Conditions d'ordonnançabilité pour un langage dirigé par le temps

T. Kloda

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)*

Présentée et soutenue le *29 septembre 2015* **par :**

TOMASZ KLODA

Conditions d'ordonnançabilité pour un langage dirigé par le temps

---

JURY

LAURENT GEORGE	Président
EMMANUEL GROLLEAU	Rapporteur
GIUSEPPE LIPARI	Rapporteur
ENRICO BINI	Examinateur
LUCA SANTINELLI	Examinateur
BRUNO D'AUSBOURG	Directeur de thèse

École doctorale et spécialité :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance
Unité de Recherche :
ONERA/DTIM
Directeur de Thèse :
Bruno d'Ausbourg
Rapporteurs :
Emmanuel Grolleau et *Giuseppe Lipari*

Remerciements

Je tiens à adresser mes remerciements les plus sincères, en tout premier lieu, à Bruno d'Ausbourg, mon directeur de thèse. Merci de m'avoir accueilli, fait confiance et soutenu. C'est grâce à ta générosité et la façon dont tu as dirigé mes travaux que j'ai pu apprendre à voir plus grand, à avoir l'esprit ouvert et le courage d'aller à contre-courant. Je te remercie du fond du cœur pour ton temps et pour tous les efforts que tu as déployés pour m'aider.

Je souhaite remercier Luca Santinelli qui s'est investi de manière très importante dans les travaux de cette thèse. Je l'ai toujours considéré comme l'un de mes encadrants et ce n'est que pour des raisons administratives que son nom n'a pas pu être cité en tant que tel. Je te remercie pour m'avoir conseillé, pour m'avoir toujours montré de nouvelles pistes à explorer et surtout pour ton regard critique sans lequel je n'aurais jamais pu voir mes défauts et ainsi les corriger.

Je voudrais remercier vivement les membres du jury pour avoir accepté de venir participer à ma soutenance de thèse et pour l'intérêt qu'ils ont porté à mes travaux de recherche. Je suis très reconnaissant à Emmanuel Grolleau et Giuseppe Lipari qui m'ont fait l'honneur d'être rapporteurs de cette thèse.

Lors de la rédaction de ce manuscrit j'ai bénéficié de l'aide de mes encadrants mais aussi de celle de deux anciens thésards, Christophe et Jean-Baptiste. Merci pour le temps que vous avez consacré à la relecture et pour vos conseils qui m'ont été très précieux.

Je tiens à exprimer ma vive gratitude aux chercheurs du Département de Traitement de l'Information et Modélisation de l'Onera Toulouse. Merci pour votre accueil, votre aide et votre soutien sans faille. Je remercie également toutes les autres personnes que j'ai pu côtoyer sur le centre Onera de Toulouse.

Merci à mes amis. L'importance de ces travaux est peu de chose comparée à l'hospitalité, l'aide et l'amitié que vous m'avez témoignés ici durant ces années.

Le plus important, je remercie ma mère, mon père et toute ma famille.

Table des matières

Chapitre 1

Introduction

1.1 Système temps réel

Un *système temps réel* est un système informatique dont le comportement déterministe ne dépend pas seulement des résultats de son calcul, mais également de l'instant auquel ces résultats sont produits [165]. Autrement dit, l'absence de réponse à un moment donné constitue un comportement indésirable pouvant mener à une défaillance du système.

Selon la manière dont leurs contraintes temporelles sont satisfaites ou pas, les systèmes temps réel peuvent être classés en deux catégories :

Les systèmes temps réel durs
: La violation d'une contrainte temporelle est inadmissible et peut conduire à une situation critique (par exemple les systèmes de commandes de vol sur un avion sont un système temps réel dur).

Les systèmes temps réel souples
: Le respect des contraintes temporelles est toujours souhaitable pour assurer un bon niveau de performance, mais le dépassement d'une contrainte dans certaines limites est toléré et ne rend pas le système inexploitable (un système de visioconférence est un exemple de système temps réel souple).

Des exemples de système temps réel peuvent être :

- un système de contrôle de réaction chimique
- un stimulateur cardiaque
- un système des commandes de vol sur un avion
- un système d'injection électronique de carburant
- un système de réservation de billet de train
- un système de guidage de missile
- un système de suivi du cours des actions en bourse
- un système de surveillance du bruit sous-marin

1.2 Caractéristiques d'un système temps réel

Réactivité Un *système réactif* [82] fournit constamment une réponse aux entrées provenant de l'environnement. Il lit ces entrées, effectue un traitement sur entrées et restitue des sorties. Il existe deux façons d'interagir ainsi avec l'environnement. Dans un *système à déclenchement événementiel* une activité est initiée suite à l'occurrence d'un événement significatif. Un *système à déclenchement temporel* observe à des instants prédéfinis l'état du processus qu'il contrôle et invoque à ces instants les activités appropriées [103].

Adaptabilité «Une seule chose est constante, permanente, c'est le changement.» La maîtrise d'un processus exige la connaissance des transformations qu'il peut être amené à subir au cours du temps. Le vol d'un avion est composé de cinq phases principales : roulage, décollage, croisière, atterrissage et roulage. Lors de chacune de ces phases, le systèmes de gestion de vol doit réaliser différentes tâches sous différentes contraintes. La *tolérance aux pannes* est la capacité d'un système à continuer de fonctionner lors de pannes de certains de ses composants et de produire des résultats corrects en mode dégradé au lieu de s'arrêter. Les systèmes *thermiquement résilients* [89] opèrent dans des conditions où la température change dynamiquement. La dissipation d'énergie thermique du système vers l'environnement dont la température atteint un niveau trop élevé peut être limitée en réduisant momentanément la performance du systèmes et son taux d'utilisation du processeur. Les systèmes nécessitant une longue autonomie peuvent collecter l'énergie de l'environnement et la stocker [45]. Leur fonctionnement ainsi que leur gestion des ressources doivent s'ajuster à la quantité d'énergie disponible.

Isolation temporelle La puissance du calcul offerte par de nouveaux processeurs permet d'héberger sur une même plateforme plusieurs applications qui devaient auparavant s'exécuter séparément sur différentes plateformes [110]. L'exécution d'une application ne doit pas être affectée par celle des autres applications s'exécutant concurremment (*temporal isolation*) [111].

Déterminisme et prédictibilité Un système déterministe est un système réagissant toujours de la même façon à une série d'événements : les mêmes entrées appliquées au système aux mêmes instants produisent les mêmes résultats. L'objectif du *calcul rapide* (*fast computing*) est de minimiser le temps moyen de réponse d'un ensemble de tâches. L'objectif du *calcul temps-réel* est de garantir que le temps de réponse de toute tâche respecte les contraintes temporelles imposées [164]. Le comportement du système temps-réel doit être déterministe dans tous les cas possibles. Le lancement de la première navette spatiale américaine s'est soldé par un échec imputable à un dysfonctionnement de son logiciel dont la probabilité d'apparition était de 1/67 [109]. Une mémoire cache enregistre temporairement des données provenant d'autres mémoires afin d'en accélérer l'accès. Quoique l'utilisation d'une mémoire cache améliore en moyenne le temps d'accès aux données, le pire temps

d'accès à une donnée s'allonge à cause du temps nécessaire à la gestion de cette mémoire lorsque la donnée demandée n'y est pas présente et doit y être transférée (*cache miss*) [38].

Indépendance entre code et matériel Un *système embarqué* est la somme d'un matériel et d'un logiciel dédiés pour réaliser ensemble une tâche précise (distributeur automatique de billets, oxymètre, photocopieur). Un logiciel indépendant du matériel peut s'exécuter sur différentes architectures matérielles. Souvent, une couche d'abstraction matérielle propose un ensemble de services pour un type de dispositif particulier par le biais desquels les programmes interagissent avec le matériel sans avoir besoin de connaître ses caractéristiques physiques. Chaque plateforme cible dispose d'une vitesse de traitement donnée (due aux capacités de processeur et d'autres périphériques) et un même programme peut ainsi avoir des comportements temporels différents en fonction de la plateforme sur laquelle il s'exécute.

1.3 Système d'exploitation temps réel

Un *système d'exploitation* assure la gestion du matériel et fournit aux applications une couche d'abstraction permettant d'exploiter ce matériel et de communiquer entre elles. Un système d'exploitation *multitâche* exécute en même temps plusieurs programmes. Le passage de l'exécution d'une tâche à l'exécution d'une autre tâche sur un processeur se fait suite à une décision de l'algorithme de *l'ordonnanceur*. Cette décision peut être prise en vue d'optimiser un certain aspect de fonctionnement du système : sa réactivité, sa consommation énergétique ou, dans le contexte temps-réel, le respect de ses contraintes temporelles. Un *protocole de communication* définit les règles selon lesquelles des processus s'exécutant concurremment échangent des données. Des processus localisés sur différentes machines communiquent à travers un *réseau informatique*. Des mécanismes de *synchronisation* contribuent à organiser et ordonner l'exécution des processus. Un processus peut nécessiter d'autres ressources que le processeur pour son exécution : des périphériques, un réseau, de la mémoire. Le *gestionnaire de mémoire* recherche et alloue de l'espace mémoire aux processus ou tâches qui en font la demande. Les processus stockent et manipulent les informations sur des mémoires de masse (disque dur, mémoire flash, disque optique) par l'intermédiaire du *système de fichiers*.

Dans un *système d'exploitation temps réel*, tous les mécanismes cités ci-dessus doivent fournir aux processus ces différents services dans des délais de temps impartis. Pour accéder aux données stockées sur un disque dur, un algorithme réorganise en général les requêtes afin de minimiser la perte de temps due aux déplacements des têtes du disque. Dans les systèmes d'exploitation temps réel, ces algorithmes servent les requêtes selon l'ordre dicté par leurs échéances temporelles en minimisant ainsi le pire temps d'accès à une donnée plutôt que le temps moyen d'accès de toutes les requêtes [117].

Il existe de nombreux systèmes d'exploitation temps réel. En voici une liste succincte.

ERIKA Enterprise est un noyau pour les systèmes temps-réel conforme au standard *OSEK/VDX* et développé sur la base de petits microcontrôleurs [67]. Comme la taille des mémoires vives dont les microcontrôleurs disposent est très restreinte, les tâches peuvent partager une même pile d'exécution (*monostack*) [68]. Des politiques d'ordonnancement à priorité fixe ainsi qu'*EDF* sont disponibles. Le noyau fonctionne sur des architectures multiprocesseurs.

RTLinux est un système d'exploitation permettant d'héberger un mini noyau temps réel et un noyau *Linux*. Celui-ci est exécuté comme une tâche de plus faible priorité et peut être préempté à tout moment par des tâches temps réel [174]. Dans cette approche, le respect de contraintes de temps réel est garanti tout en conservant les services de plus haut niveau offerts par Linux.

Xenomai utilise un hyperviseur (*Adeos*) afin de partager les ressources matérielles et dispatcher les interruptions entre tâches temps réel et tâches normales de *Linux*.

QNX est un micro noyau temps réel dont toutes les fonctionnalités (système de fichier, pilotes, réseau) sont exécutées sous forme de tâches et sont activées ou désactivées selon les besoins de l'utilisateur [90].

VxWorks est le système d'exploitation temps réel multitâche le plus largement déployé au monde. Afin de faciliter le processus de développement et de certification un modèle de programmation supportant le partitionnement spatial et temporel est proposé. Un débogage à distance permet de trouver des erreurs dans le logiciel et de les corriger à la volée (lors de la mission de *Mars Pathfinder* un bug dans le système du rover se trouvant déjà sur Mars fut corrigé depuis le centre de contrôle [12]).

1.4 Langages pour la programmation de systèmes temps réel

Un *programme informatique* est une séquence d'instructions réalisant un objectif donné [163]. Son fonctionnement est décrit par des expressions, formulées dans un *langage de programmation*, qui sont ensuite transformées en code exécutable sur une machine. Par le biais de son *interface de programmation*, le système d'exploitation peut rendre des services (manipulation du matériel, communication entre les processus) accessibles depuis un langage de programmation.

Voici quelques exemples de langages de programmation temps réel :

ADA est un langage de programmation structuré, impératif et orienté objet. Le langage permet de définir des tâches et de les munir de priorités. Plusieurs politiques d'ordonnancement, mécanismes de synchronisation et de gestion du temps sont proposés.

Java Temps Réel étend la Machine Virtuelle Java avec des fonctionnalités temps réel. Le récupérateur de mémoire (*Garbage Collector*) de la Machine Virtuelle Java a un comportement non indéterministe en préemptant les tâches à des instants imprévisibles. Dans la version temps réel, le récupérateur de mémoire s'exécute de manière prévisible. Les classes permettant de créer les tâches temps réel et les ordonnancer sont fournies dans l'interface de programmation.

API de Xenomai est l'interface de programmation en *C* pour *Xenomai* (voir section précédente). Elle facilite la création des tâches, l'allocation des priorités, la synchronisation et le partage de données.

AADL (*Architecture Analysis and Design Language*) [61] est un langage de description d'architecture pour les systèmes temps réel. Il permet de spécifier une plateforme matérielle ainsi que le logiciel destiné à s'exécuter sur celle-ci. La modélisation se focalise sur les interactions entre différents composants du système. Les composants interagissent entre eux par l'intermédiaire d'interfaces. Un *thread* représente l'exécution séquentielle d'un code. Plusieurs threads peuvent s'exécuter en parallèle. Leur exécution est décrite par des paramètres (pire temps d'exécution, période), des schémas d'activation (périodique, apériodique, sporadique), la taille de la mémoire ou le processeur d'exécution. Le framework intermédiaire *MoSaRT* [125, 124, 126] permet de coupler le langage *AADL*, ainsi que d'autres langages de modélisation des systèmes temps-réel, avec les outils d'analyse d'ordonnançabilité.

les langages synchrones sont conçus pour la programmation des *systèmes réactifs*. Le système réactif idéal fournit les résultats de son calcul en même temps qu'il reçoit ses entrées (*hypothèse de synchronisme*) : temps de calcul et temps de réaction sont nuls. Si on considère qu'un système réactif non-idéal produit ses résultats suffisamment vite après avoir lu ses entrées et avant d'en recevoir de nouvelles, ce système peut être représenté par le système idéal. L'analyse, la conception et la programmation basées sur un modèle du système réactif idéal sont plus faciles qu'avec un modèle de système traditionnel. Le temps réel est remplacé par un temps logique défini comme une séquence d'instants. A chaque instant successif une réaction du système se produit. Plusieurs langages synchrones ont été proposés : *Esterel* [30, 31], *Lustre* [43, 81, 29], *Prelude* [65, 66].

les langages dirigés par le temps permettent de spécifier la description du comportement temporel d'une application séparément de la description de son comportement fonctionnel. Les applications sont donc spécifiées par deux descriptions : la description temporelle exprimée dans un langage temporel et la description fonctionnelle exprimée dans un langage de programmation traditionnel (par exem-

ple C). La description temporelle précise exactement les instants auxquels communication et traitement sont initiés. La communication est considérée comme une opération immédiate de durée nulle tandis que la durée de traitement est non nulle et doit être toujours spécifiée. Un traitement est associé à l'appel d'une fonction définie dans la description fonctionnelle et son exécution est représentée par une *tâche*. A la date de son activation, une tâche lit ses entrées, elle s'exécute ensuite pendant le temps de calcul qui lui est alloué (ne mettant en jeu aucune interaction avec aucune autre tâche) et elle écrit à sa terminaison les résultats du calcul qui deviennent disponibles pour d'autres tâches du système. L'idée consistant à concevoir un système en séparant clairement fonctionnalité et gestion du temps a été proposée pour la première fois dans le langage *Giotto* [86, 83, 85] et ensuite développée dans les langages tels que *xGiotto* [72, 150], *Timing Definition Language* (*TDL*) [140, 59, 1, 60], *Hierarchical Timing Language* (*HTL*) [84].

1.5 Problématique

Parmi les langages cités, il semble que les langages dirigés par le temps puissent satisfaire en grande partie les exigences imposées par la conception d'un système temps réel tel que décrit en section 1.2 :

Réactivité Les langages dirigés par le temps permettent de décrire comment le système interagit avec son environnement par le biais des *capteurs* et des *actuateurs*. Les instants auxquels leurs valeurs sont mises à jour sont précisément définis.

Adaptabilité Les langages dirigés par le temps peuvent permettre de décrire comment un système évolue en passant d'un mode de fonctionnement à l'autre lorsqu'une condition donnée est vérifiée par un *commutateur de mode*. Chaque mode de fonctionnement est caractérisé par l'ensemble des tâches qu'il exécute périodiquement.

Isolation temporelle Les langages dirigés par le temps permettent, grâce à la notion de *modules*, de décomposer un système en plusieurs parties autonomes exécutées de façon distribuée. L'exécution de chaque module est indépendante des exécutions des autres modules.

Déterminisme et prédictibilité Tout instant de lancement et de terminaison d'une tâche ainsi que tout instant d'échange de données sont complètement définis à l'aide des langages dirigés par le temps.

Indépendance entre code et matériel Les langages dirigés par le temps imposent que le comportement observable d'un système soit le même sur toutes les plateformes d'exécution. Le compilateur des langages doit assurer cet impératif. Les instants de lecture d'entrées et d'écriture de résultats sont fixés lors de la conception. Les résultat

de calcul ne deviennent accessibles qu'à ces instants quel que soit l'instant effectif de la fin d'exécution.

Un micro-noyau mettant en œuvre les mécanismes et les services nécessaires au contrôle de l'exécution d'applicatifs temps réel dirigés par le temps a été développé au sein du département de recherche *DTIM* de *l'ONERA*. Ce micro-noyau temps réel s'appelle *KBR* et est décrit plus en détail dans la section suivante. Une machine virtuelle intégrée à *KBR* contrôle l'exécution des tâches dirigées par le temps. Le schéma de cette exécution est exprimé par un *code de contrôle* représentant une suite d'actions à opérer à des échéances données. Ce code est le résultat de la compilation du programme temporel de l'application rédigé en langage dirigé par le temps. Ce code est généré à partir des programmes rédigés dans le langage proposé comme une extension du *Timing Definition Language* (*TDL*). La définition de ce langage, appelé *Extended - Timing Definition Language* (*E-TDL*) [18], est présentée en section 1.5.2.

La principale question abordée dans cette thèse est celle de la *faisabilité* des programmes *E-TDL* [100]. Plusieurs tâches décrites dans un programme *E-TDL* s'exécutent concurremment sur un processeur. L'exécution complète de chacune d'elles nécessite un temps de processeur donné et doit se terminer dans les échéances imposées. Il est alors important de s'assurer que les ressources de processeur sont suffisantes afin que toutes les tâches respectent leurs échéances.

1.5.1 Noyau temps réel KBR

KBR est un micro-noyau implantant les mécanismes et services nécessaires au contrôle de l'exécution d'applicatifs temps réel *dirigés par le temps*.

Jusqu'à présent, l'exécution des applications décrites en langages dirigés par le temps était fondée sur des systèmes d'exploitation pré-existants [58, 87]. Ces systèmes, basés sur le paradigme événementiel, nécessitent une couche additionnelle afin de permettre l'interprétation et l'exécution correctes des programmes dirigés par le temps. L'introduction de cette couche entre applications et système opératoire pénalise la performance du système et dégrade la précision du contrôle temporel. Le micro-noyau temps réel *KBR* élimine la nécessité d'une couche intermédiaire par l'implantation, au sein d'un système opératoire, des structures et des mécanismes de base permettant l'exécution et le contrôle des tâches dirigées par le temps.

L'exécution dirigée par le temps des tâches, sur *KBR*, repose principalement sur deux mécanismes introduits et implantés au sein du noyau :

- le mécanisme de traitement des interruptions horloge (*Intel 8524*)
- le mécanisme de gestion d'échéances temporelles

La machine virtuelle chargée d'interpréter le code de contrôle de l'exécution des tâches s'appuie sur ces mécanismes de synchronisation pour interpréter le code de contrôle généré par le compilateur à partir de la spécification temporelle formulée dans le langage *E-TDL*.

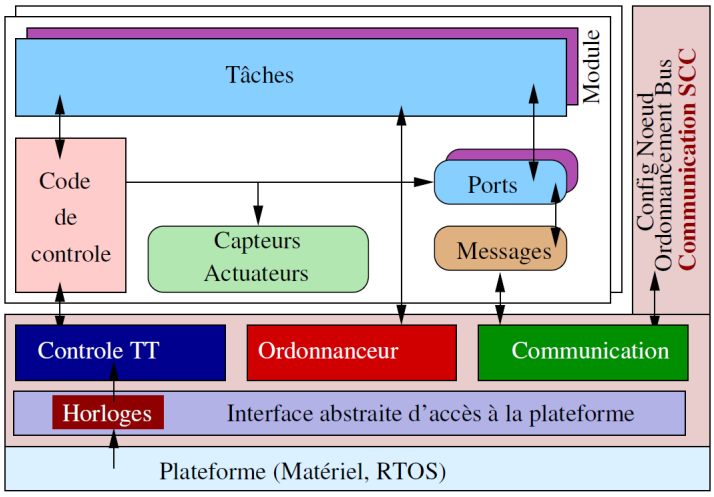


FIGURE 1.1 – L'architecture de *KBR*

Les instructions de ce code de contrôle désignent une action à opérer à une échéance donnée. A chacune des échéances est associée l'adresse de la fonction qui doit être activée lorsque la date courante atteint la date d'échéance fixée. Le mécanisme de gestion des échéances permet donc de mémoriser au sein du système l'organisation temporelle de blocs d'actions à opérer à dates données (voir Figure 1.2). Le déclenchement de ces actions est effectué à l'échéance qui leur est fixée par le mécanismes de gestion de l'interruption horloge.

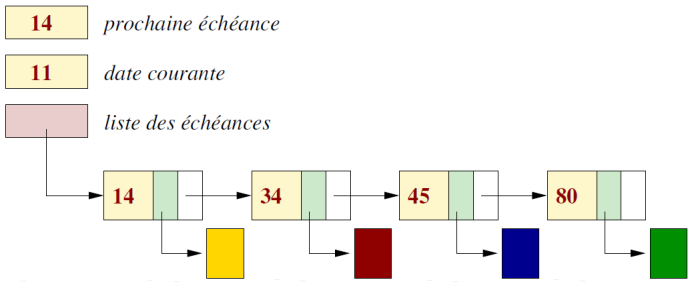


FIGURE 1.2 – Listes d'échéances dans *KBR*

Un autre aspect important pour assurer le fonctionnement déterministe du système dirigé par le temps consiste à établir des protocoles de communication entre les tâches. Les dates auxquelles celles-ci mettent à jour leurs sorties et lisent leurs entrées sont strictement définies et doivent être impérativement respectées. Dans *KBR* des mécanismes appropriés sont implantés (*ports*) pour garantir des échanges de données corrects concernant leur

exactitude temporelle ainsi que l'allocation de mémoire et la gestion d'accès pour des données partagées. *KBR* offre également des mécanismes de base pour la gestion de la mémoire physique et virtuelle.

1.5.2 E-TDL

Les langages dirigés par le temps se fondent sur la notion de *Temps d'Exécution Logique* (*TEL*) qui rend possible de décrire le comportement temporel d'une application en faisant abstraction de son implantation spécifique pour une plateforme donnée. D'un point de vue

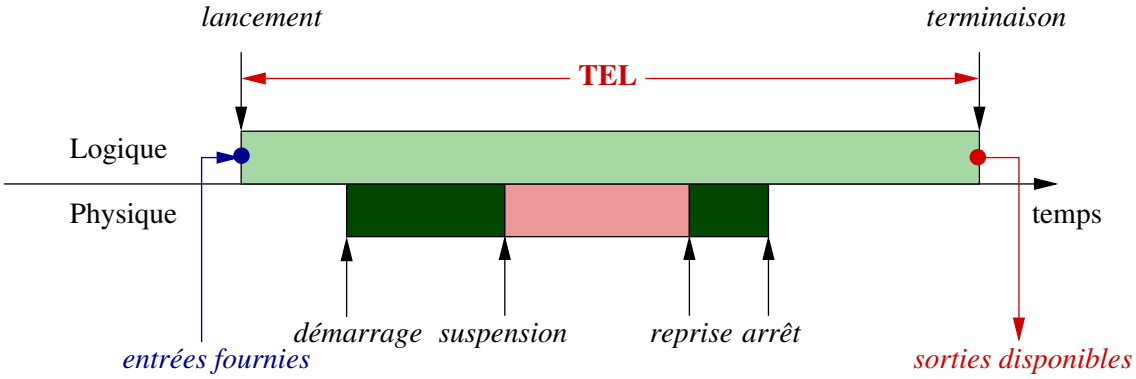


FIGURE 1.3 – Temps d'Exécution Logique

logique, et en se fondant sur le paradigme de déclenchement temporel du système, une tâche lit ses entrées à sa date de lancement, elle s'exécute jusqu'à sa date de terminaison et rend les résultats disponibles à cet instant précis. D'autre part, d'un point de vue physique, une tâche peut démarrer après sa date de lancement, elle peut être préemptée ou suspendue au cours de son exécution et achever son travail avant sa date de terminaison. La conformité de ces deux perspectives est assurée par le fait que les entrées sont lues au moment de lancement, même si la tâche retarde le début de son exécution, et les résultats sont fournis exactement à la date de terminaison quelle que soit la date physique de la fin d'exécution. Ceci veut dire que cette approche détermine précisément quelle valeur est en usage à quel instant et apporte ainsi au système son caractère déterministe.

La principale entité du langage *E-TDL* est la tâche dotée de son code fonctionnel qui s'exécute périodiquement. Les tâches interagissent entre elles et avec l'environnement par l'intermédiaire des *ports*. Dans les langages dirigés par le temps déjà existants (*Giotto*, *TDL*), la période de la tâche est toujours égale à son *Temps d'Exécution Logique*. Le langage *E-TDL* étend ce modèle de tâche principalement en différenciant le *Temps d'Exécution Logique* (*TEL*) d'une tâche de sa période. Par l'introduction de la notion de *phase*, une tâche peut lire ses entrées et commencer son exécution après un certain délai suivant le

début de la période. Si son *Temps d'Exécution Logique* est inférieur à sa période, la tâche peut terminer son exécution et rendre les résultats de son calcul disponibles avant la fin de la période.

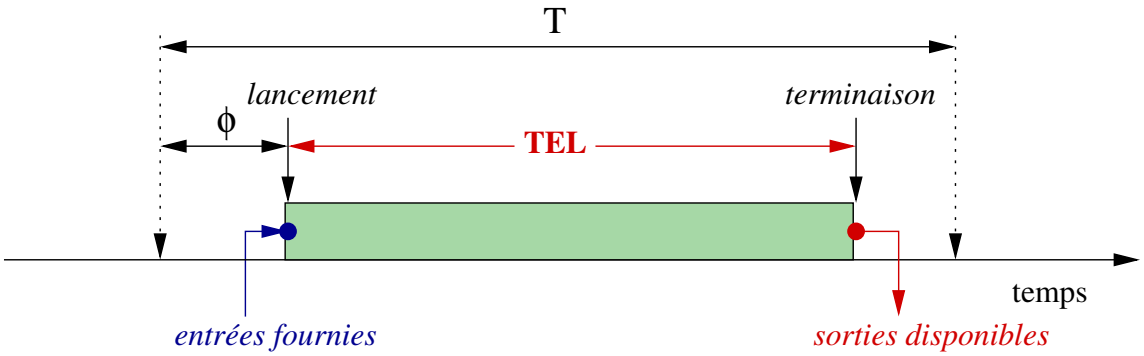


FIGURE 1.4 – Temps d'Exécution Logique d'une tâche en E-TDL

Définition 1.1 (**Tâche**). *Une tâche E-TDL* $\tau = (\Phi, C, TEL, T)$ *est associée à l'exécution d'un calcul et est caractérisée par les paramètres suivantes :*

Φ	*est la phase (offset) avec laquelle la tâche commence son exécution,*
C	*est son pire temps d'exécution (wcet) sur une machine donnée,*
TEL	*est son Temps d'Exécution Logique,*
T	*est sa période.*

Un *mode* en E-TDL représente un état de fonctionnement du programme. Différentes tâches peuvent être regroupées au sein de modes et peuvent coopérer ensemble à la réalisation d'un certain travail. Des tâches de même mode s'exécutent concurremment.

Définition 1.2 (**Mode**). *Un mode* m *dans E-TDL est déterminé par sa période* $T[m]$ *et un ensemble de tâches* $\tau[m]$ *exécutées dans ce mode. La période* $T[m]$ *est un multiple commun des périodes des tâches* $\tau[m]$ *:*

$$\forall(\Phi, C, TEL, T) \in \tau[m] \ : \ \exists n \in \mathbb{N}_+, \, n \cdot T = T[m] \tag{1.1}$$

Au fil du temps, un mode déclenche des activités comme des invocations de tâches, des mises à jours d'actuateurs, des commutations de modes. Ces activations sont déclenchées à des instants bien précis et bien définis par rapport à l'instant de début de mode.

Définition 1.3 (**Temps de mode**). *Le temps de mode* δ *est le temps relatif au mode écoulé depuis la plus récente activation de ce mode. Si* m *est un mode courant, son temps de mode* δ *satisfait l'inégalité suivante :*

$$0 \leq \delta < T[m] \tag{1.2}$$

Le système peut évoluer en passant d'un mode à l'autre lorsqu'une condition donnée est vérifiée par *un commutateur de mode*. La commutation de mode est effectuée en un temps négligeable. Elle peut être déclarée au sein d'un mode à une fréquence donnée pourvu qu'elle n'interrompe l'exécution logique d'aucune des tâches du mode courant.

Définition 1.4 (**Commutation de mode**). *Une commutation de mode est décrite par un triplet* $(m, m', T_{sw}(m, m'))$ *tel que :*

m	*est le mode courant,*
m'	*est le mode qui sera exécuté si la condition de changement de mode est vérifiée,*
$T_{sw}(m, m')$	*est la période avec laquelle la condition de changement de mode est évaluée dans le mode* m *telle que la relation suivante est satisfaite :*

$$\forall(\Phi, C, TEL, T) \in \tau[m] \ : \ T_{sw}(m, m') \bmod T = 0 \qquad (1.3)$$

La Formule 1.3 garantit que le changement de mode n'interrompt l'exécution logique d'aucune des tâches du mode courant : toute tâche est arrêtée à la fin de sa période. Si la condition de changement de mode n'est pas vérifiée, l'exécution du mode courant continue.

Définition 1.5 (**Prochains modes**). *Soit un temps de mode* δ *dans un mode* m*. La fonction* $prochains\_modes(m, \delta)$ *retourne l'ensemble des modes pouvant, toutes conditions confondues, commencer immédiatement après* δ*.*

Modes, tâches, ports d'entrée et de sortie forment les *modules*. Les modules sont les entités autonomes permettant de décomposer l'application et de l'exécuter de façon distribuée. Un programme *E-TDL* peut être composé de plusieurs modules s'exécutant en même temps sur le même processeur. Chaque module exécute un mode à la fois.

Définition 1.6 (**Module**). *Un module* M *exécute à un instant donné un de ses modes pris dans l'ensemble* $Modes[M]$*. Le mode initial exécuté lors du démarrage du système est désigné comme* $Init[M] \in Modes[M]$*.*

Définition 1.7 (**Etat de module**). *L'état d'un module à un instant donné est défini par le couple* (m, δ) *où* δ *est le temps de mode et où* m *est le mode en train de s'exécuter à cet instant.*

Exemple 1.1. *Soit un régulateur qui délivre un signal de commande calculé à partir de deux grandeurs physiques mesurées par des capteurs. Les valeurs des mesures sont transmises au régulateur par un réseau du type Time Division Multiple Access (TDMA) de sorte que :*

- *la valeur courante du premier capteur est transmise dans les créneaux :*

$$[0, 10], [40, 50], [80, 90], \ldots$$

- *la valeur courante du second capteur est transmise dans les créneaux :*

$$[20, 30], [60, 70], [100, 110], \ldots$$

Le signal de commande doit être mis à jour toutes les 40 unités de temps. L'algorithme du régulateur peut être décomposé en trois tâches :

τ_1	*lecture de la valeur du premier capteur*
τ_2	*lecture de la valeur du second capteur*
τ_3	*calcul du signal de commande à partir des valeurs lues depuis les capteurs*

Le Listing 1.1 présente un exemple de programme rédigé en E-TDL pour le régulateur. Les fonctions TDMA_read et compute_control doivent être définies dans un autre fichier associé au programme. Le module Regulateur a deux capteurs (leurs identifiants dans le réseau) et un actuateur (valeur du signal de commande). Au démarrage du système le module s'exécute une seule fois dans son mode initial. Les valeurs du premier et du second capteur sont lues, respectivement, par la tâche τ_1 et par la tâche τ_2. Ces tâches, déclenchées dans les intervalles appropriés, établissent la communication par le réseau TDMA avec les capteurs et reçoivent leurs valeurs courantes. Le pire temps (wcet) de cette opération s'élève à 10 unités de temps. A la fin de la période du mode initial (40 unités de temps), un changement de mode est effectué. Le module passe désormais à l'exécution d'un mode normal (dont la période est également de 40 unités de temps). Dans ce mode, une nouvelle tâche τ_3 est ajoutée. Elle calcule, à partir des valeurs des capteurs mises à jour par les tâches τ_1 et τ_2, la valeur du signal de commande. Son pire temps d'exécution est égal à 20 unités de temps.

Listing 1.1 – Un exemple de programme en *E-TDL* pour le régulateur de l'Exemple 1.1

```
module Regulateur {
  sensor
    int          capteur1;
    int          capteur2;
  actuator
    int          commande:=0;
  task t1 [wcet=10] {
      input
        int      capteur_adresse;
      output
        int      capteur_valeur;
      calls
        TDMA_read(capteur_adresse, capteur_valeur);
  }
  task t2 [wcet=10] {
      input
```

```
        int       capteur_adresse;
      output
        int       capteur_valeur;
      calls
        TDMA_read(capteur_adresse, capteur_valeur);
  }
  task t3 [wcet=20] {
      input
        int       capteur_valeur1;
        int       capteur_valeur2;
      output
        int       u;
      calls
        compute_control(capteur_valeur1, capteur_valeur2, u);
  }
  init mode initial [period=40] {
      task
        [freq=1][tel=10][phase=0]          t1(capteur1);
        [freq=1][tel=10][phase=20]         t2(capteur2);
      mode
        [freq=1] if True then normal
  }
  mode normal [period=40] {
      task
        [freq=1][tel=10][phase=0]          t1(capteur1);
        [freq=1][tel=10][phase=20]         t2(capteur2);
        [freq=1][tel=40][phase=0]          t3(t1.output, t2.output);
      actuator
        [freq=1] commande:=t3.output
  }

}
```

Compte tenu des définitions introduites dans cette section, le module Regulateur est décrit comme suit :

$$Modes[Regulateur] = \{initial, normal\},\ Init[Regulateur] = initial$$
$$T[initial] = T[normal] = T_{sw}(initial, normal) = 40$$
$$\tau[initial] = \{\tau_1, \tau_2\},\ \tau[normal] = \{\tau_1, \tau_2, \tau_3\}$$
$$\tau_1 = (0, 10, 10, 40),\ \tau_2 = (20, 10, 10, 40),\ \tau_3 = (0, 20, 40, 40)$$

La Figure 1.5 représente l'exécution du module Régulateur pendant 120 unités de temps à partir de son démarrage. Dans l'intervalle $[0, 40]$ *les tâche* τ_1 *et* τ_2 *sont exécutées par le mode initial. A l'instant 40, le mode initial est remplacé par le mode normal et la tâche* τ_3 *intègre les deux premières tâches.*

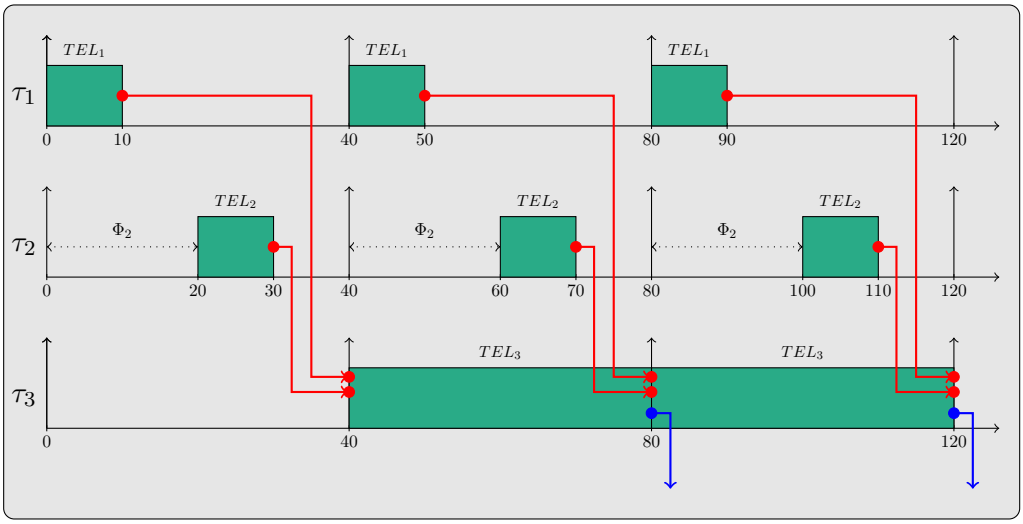


FIGURE 1.5 – L'Exécution du module Régulateur de l'Exemple 1.1

Remarque 1.1. *Dans les langages dirigés par le temps classique (Giotto, TDL) le Temps d'Exécution Logique d'une tâche τ est égal à sa période ($\Phi = 0, TEL = T$). En E-TDL une tâche τ peut comporter un décalage Φ et un $TEL \leq T$, $\tau = (\Phi, C, TEL, T)$. Pour obtenir dans les langages classiques une description équivalente à celle de τ dans E-TDL, il faudrait définir les trois modes successifs suivants :*

$$
\begin{aligned}
m_1 \qquad & T[m_1] = \Phi,\ \tau[m_1] = \emptyset \\
m_2 \qquad & T[m_2] = TEL,\ \tau[m_2] = \{\tau = (0, C, TEL, TEL)\} \\
m_3 \qquad & T[m_3] = T - TEL - \Phi,\ \tau[m_3] = \emptyset
\end{aligned}
$$

L'exécution d'une période du mode m_1 est suivie de l'exécution d'une période du mode m_2 qui est ensuite suivie de l'exécution d'une période du mode m_3 correspond à une exécution de la tâche E-TDL τ. Pourtant, dans l'analyse d'ordonnançabilité des programmes TDL *proposée en [59], le redémarrage de l'exécution du mode est toujours considéré. Cette analyse prendrait en compte l'exécution continue du mode m_2, nécessitant C unités de traitement processeur, dans tout intervalle de durée $n \cdot TEL$ pour $n \in \mathbb{N}$. Dans le cas d'E-TDL, le schéma d'exécution de la tâche τ est naturellement et explicitement défini. Dans l'analyse qui sera proposée pour E-TDL, la demande de temps processeur dans les mêmes intervalles est estimée plus précisément et sans surestimation.*

1.6 Contributions

Les principales contributions de cette thèse sont :

- Proposition de conditions d'ordonnançabilité [97] sous *EDF* de systèmes temps réel décrits en *E-TDL*, exécutés sur des architectures monoprocesseur et en présence d'autres composants utilisant également les ressources processeur.
- Réalisation d'un outil implantant une analyse d'ordonnançabilité de tels systèmes par la vérification de ces conditions.

L'outil est destiné à être intégré au sein du compilateur *d'E-TDL*. Vérifier l'ordonnançabilité des applications en *E-TDL* permet de garantir que le schéma du contrôle temporel de leur exécution, généré par le compilateur, est conforme au modèle d'exécution dénoté dans leur énoncé des exigences temporelles. Le programmeur des applicatifs embarqués temps réel peut alors bénéficier d'un cadre facilitant leur conception et leur développement. Grâce à la séparation claire entre la part fonctionnelle des applications et la spécification des exigences temporelles induite dans le concept de *Temps d'Exécution Logique*, le programmeur se voit déchargé de la programmation de la gestion du temps pour se concentrer sur les fonctions de son application. Les applications embarqués dont le comportement temporel est spécifié, de cette façon, indépendamment de sa fonction et de sa plateforme d'exécution sont plus aisées à modifier, tester, certifier et porter vers d'autres architectures cibles.

Chapitre 2

Ordonnancement temps réel

Un système temps réel est constitué par plusieurs tâches qui s'exécutent sur un ensemble de processeurs disponibles et partagent entre elles certaines ressources. Les exécutions des tâches doivent respecter les contraintes temporelles imposées par la spécificité du processus dont l'évolution est contrôlée par le système. La durée totale qu'une tâche nécessite pour achever entièrement son travail est non-négligeable et dépend du jeu d'instructions que celle-ci doit exécuter. Une seule tâche peut s'exécuter sur chaque processeur à tout moment. L'accès simultané à des ressources partagées peut être limité à un nombre particulier de tâches. Le fonctionnement correct du système dépend, par conséquent, de l'ordre dans lequel les tâches sont exécutées. L'ordonnancement temps réel cherche à construire des algorithmes qui établissent cet ordre et à trouver des méthodes pour vérifier que cet ordre garantit effectivement le respect de toutes les contraintes.

2.1 Tâche temps réel

Une *tâche temps réel* est l'entité de base traitée par le système temps réel. Elle représente les calculs et les opérations exécutés en série sur un processeur [147, 38].

Une exécution concrète d'une tâche sur un processeur est appelée une instance. Selon la corrélation entre les dates d'activation de leurs instances successives, les tâches sont différenciées de la manière suivante.

Une tâches périodique est exécutée cycliquement et les dates d'activation de ses instances suivantes sont séparées par un intervalle de temps constant appelé *période*.

Une tâche apériodique ne montre aucune relation ni régularité entre les dates d'activation des instances consécutives.

Une tâche sporadique est caractérisée par un intervalle de temps minimal, déterminé d'avance, qui sépare les dates d'activation de ses instances subséquentes.

Une *tâche temps réel* τ_i est caractérisée par les paramètres suivants :

Temps d'exécution maximum - C_i
la durée maximale nécessaire pour terminer l'exécution non interrompue de cette tâche

Date d'activation - $r_{i,j}$
l'instant à partir duquel sa j^{ieme} instance $\tau_{i,j}$ est prête à commencer son exécution

Phase - ϕ_i
l'instant à partir duquel sa première instance $\tau_{i,1}$ est prête à commencer son exécution

Période - T_i
l'intervalle de temps qui sépare les dates d'activation successives de la tâche τ_i si celle-ci est périodique ; $r_{i,j} = \phi_i + (j-1) \cdot T_i$

Date de début d'exécution - $s_{i,j}$
le moment à partir duquel sa j^{ieme} instance $\tau_{i,j}$ commence son exécution ; $s_{i,j} \geq r_{i,j}$

Date de fin d'exécution - $f_{i,j}$
le moment auquel sa j^{ieme} instance $\tau_{i,j}$ finit son exécution

Temps de réponse $R_{i,j}$
le temps entre *la date d'activation* et *la date de fin d'exécution* de sa j^{ieme} instance $\tau_{i,j}$; $R_{i,j} = f_{i,j} - r_{i,j}$

Date d'échéance relative - D_i
l'intervalle de temps, à partir de sa date d'activation, dans lequel une instance de tâche doit se terminer

Date d'échéance absolue - $d_{i,j}$
l'instant à partir duquel sa j^{ieme} instance $\tau_{i,j}$ doit être terminée ; si la tâche τ_i est périodique : $d_{i,j} = r_{i,j} + D_i$

Temps de retard - $L_{i,j}$
la différence entre *la date de fin d'exécution* et *la date d'échéance absolue* de sa j^{ieme} instance $\tau_{i,j}$; $L_{i,j} = f_{i,j} - d_{i,j}$

Laxité - $X_{i,j}$
l'intervalle de temps maximal duquel la date de début d'exécution de la j^{ieme} instance de la tâche τ_i peut être retardée sans dépassement de son échéance

2.2 Ordonnancement

L'objectif d'un algorithme d'ordonnancement est de résoudre le problème caractérisé par trois ensembles : un ensemble de tâches $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$, un ensemble de processeurs $P = \{P_1, P_2, \ldots, P_m\}$ et un ensemble de ressources $R = \{R_1, R_2, \ldots, R_s\}$. L'algorithme d'ordonnancement, à chaque instant t de fonctionnement du système, affecte les processeurs appartenant à P et, si nécessaire, les ressources appartenant à R aux tâches dans Γ afin de permettre à toutes les tâches d'achever leur travail dans le respect des contraintes imposées.

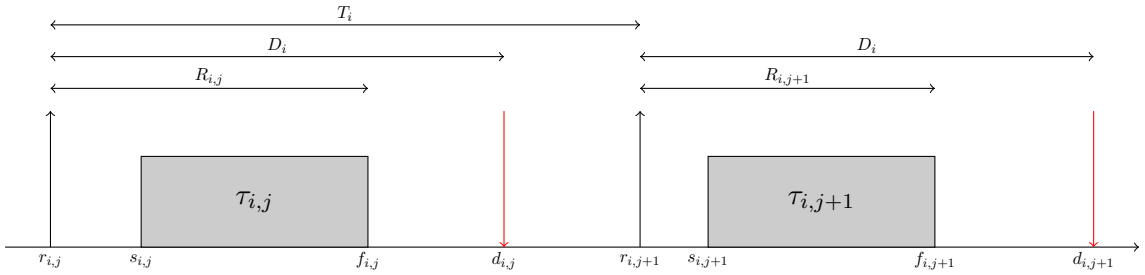


FIGURE 2.1 – Modèle d'une tâche temps réel périodique

A un instant donné, chaque tâche peut être exécutée sur un processeur seulement et chaque processeur est capable d'exécuter au plus une tâche à la fois [35].

La validation d'un système temps réel consiste, pour son aspect temporel, à déterminer si l'algorithme d'ordonnancement utilisé garantit que les échéances de toutes les tâches seront, sous les contraintes données, respectées à tout instant. Si cela est vrai, le système est dit *ordonnançable* par cet algorithme. Le système de tâche est dit *faisable* s'il existe un algorithme quelconque qui le rende ordonnançable. L'algorithme d'ordonnancement est *optimal* dans une classe s'il n'existe aucun autre algorithme de sa classe qui rende le système ordonnançable tandis que celui-ci ne le fait pas. Si l'algorithme est optimal et qu'il ne peut ordonnancer les tâches du système sous l'ensemble des contraintes présentes, alors aucun algorithme de sa classe ne pourra le faire.

Les algorithmes d'ordonnancement sont classés en plusieurs catégories en fonction du type de contraintes et des caractéristiques du système.

Hors ligne/en ligne Les algorithmes *hors ligne* fixent toutes les décisions avant le lancement de l'application. Cela nécessite que les paramètres de toutes les tâches soient connus *a priori*. La séquence d'ordonnancement est construite lors de la compilation. Elle est enregistrée dans un tableau associant des unités de temps aux tâches sélectionnées pour l'exécution. Ainsi, l'ordonnanceur minimise son surcoût au détriment de la flexibilité.
Dans les algorithmes dits *en ligne* le choix d'une tâche à démarrer s'effectue au cours de l'exécution de l'application. Au fur et à mesure de l'évolution de l'état du système, l'algorithme choisit chaque fois qu'il est activé une tâche parmi celles prêtes à être exécutées. Le problème devient ici du coup plus complexe et sa résolution plus coûteuse en temps. L'algorithme s'adapte cependant à tout changement intervenant dans l'ensemble des tâches (par exemple l'insertion d'une nouvelle tâche). L'approche *en ligne* n'exige aucune connaissance des paramètres des tâches lors de la compilation du programme.

Priorité L'algorithme d'ordonnancement attribue des priorités aux tâches de manière à établir une hiérarchie parmi les tâches prêtes à être exécutées et de manière à décider

aisément à quelle tâche attribuer le processeur. La tâche ayant la priorité la plus élevée est désignée pour l'exécution. Les algorithmes à *priorité fixe* affectent à chaque tâche une priorité qui reste constante tout au long de la vie du système (par ex. *Rate Monotonic*). A l'inverse, les algorithmes à *priorité dynamique* assignent des priorités qui peuvent être modifiées à tout moment de l'exécution. Parmi ces derniers on distingue ceux qui affectent une priorité fixe et constante à chacune des instances d'exécution de la tâche (par ex. *Earliest Deadline First*) et ceux qui autorisent une évolution de cette priorité lors de l'exécution de l'instance (par ex. *Least-Slack Time First*).

Monoprocesseur/multiprocesseur Un algorithme est un algorithme *monoprocesseur* si l'ensemble de processeurs ne compte qu'un élément. S'il s'applique à plusieurs processeurs, l'algorithme est dit *multiprocesseur*.
Pour ces derniers, on distingue deux principales approches :

L'ordonnancement par partitionnement exige qu'une tâche soit exécutée toujours sur le même processeur. La migration d'une tâche d'un processeur à un autre est interdite.

L'ordonnancement global permet la migration d'une tâche entre les différents processeurs. La migration peut cependant être restreinte au niveau d'une instance de la tâche et permettre ou non la poursuite de l'exécution de cette instance sur un autre processeur que celui sur lequel elle a démarré.

Préemption Les algorithmes peuvent aussi être caractérisés par leur capacité à préempter les tâches. Un algorithme *préemptif* est capable de suspendre l'exécution d'une tâche et de commencer l'exécution d'une autre. Au contraire, un algorithme *non-préemptif* ne possède pas cette propriété.

Conservatif Un algorithme *conservatif* sélectionne à chaque instant une tâche à exécuter. Il ne laisse jamais un processeur inoccupé (excepté si l'ensemble de tâches prêtes à exécuter est déjà vide).

Partage des ressources La répartition des ressources (telles qu'une zone de mémoire, des variables ou un périphérique) entre les tâches et la nature de cette répartition sont des facteurs dont l'algorithme d'ordonnancement doit tenir compte. L'accessibilité d'une ressource peut être limitée à un nombre déterminé de tâches. Citons, à titre d'exemple, deux problèmes classiques de partage des ressources.

L'inversion de priorités est une situation où une tâche τ est empêchée de s'exécuter par des tâches moins prioritaires dont l'une d'entre elles possède une ressource dont la tâche τ a besoin.

L'interblocage peut avoir lieu lorsque deux tâches concurrentes s'attendent mutuellement, chacune en attendant que l'autre libère une ressource.

Plusieurs protocoles et méthodes d'analyse ont été définis pour résoudre le problème du partage des ressources [152, 20, 21].

Précédence Une autre forme de dépendance entre tâches s'exprime par une contrainte de *précédence*. Elle impose un ordre partiel sur l'exécution des tâches appartenant au même groupe. Les contraintes de précédence sont définies par un graphe acyclique orienté où l'ensemble des sommets représente l'ensemble des tâches et pour tous sommets i et j l'arc $i - j$ implique que la tâche i doit être impérativement terminée avant que la tâche j ne commence son exécution.

Ensemble de tâche synchrones/asynchrones S'il existe un instant t tel que toutes les instances de toutes les tâches du système peuvent démarrer simultanément à t, cet ensemble de tâches est appelé *synchrone*. Dans le cas contraire, si un tel instant n'existe pas, l'ensemble de tâches est dit *asynchrone*.

Changement de mode En réponse à l'évolution des conditions dans lesquelles il fonctionne, le système peut demander à modifier l'ensemble de tâches en cours d'exécution ou même à le remplacer par un autre mieux adapté à ces nouvelles conditions. Un *mode*, du point de vue de l'ordonnancement, est associé à un groupe de tâches dont l'exécution dépend de l'état dans lequel se trouvent le système et son environnement. Les règles affectant le *changement de mode*, c'est-à-dire, la façon dont les modes peuvent se succéder, sont définies dans des protocoles de changement de mode. Le changement de mode, engendrant parfois une phase transitoire de surcharge du processeur, fait l'objet d'analyses diverses qui seront plus détaillées et discutées dans les sections suivantes.

2.3 Principaux algorithmes

Le problème de l'ordonnancement peut prendre, selon les types des contraintes rencontrées, des formes très variées. De nombreux algorithmes d'ordonnancement ont été proposés et étudiés dans la littérature. Leur abondance ne permet pas de les traiter tous dans ce mémoire. Une attention plus particulière sera ainsi accordée à ceux qui sont les plus connus et les plus usités en contexte monoprocesseur et préemptif.

2.3.1 Rate Monotonic

L'algorithme *Rate Monotonic* (*RM*) est un algorithme préemptif à priorité fixée pour les tâches ayant leurs dates d'échéance égales à la durée de leurs périodes ($D_i = T_i$). Les priorités sont assignées en donnant la préférence aux tâches caractérisées par les périodes les plus courtes. L'algorithme Rate Monotonic a été introduit et prouvé optimal dans la classe des algorithmes à priorité fixée par Liu et Layland en 1973 [112].

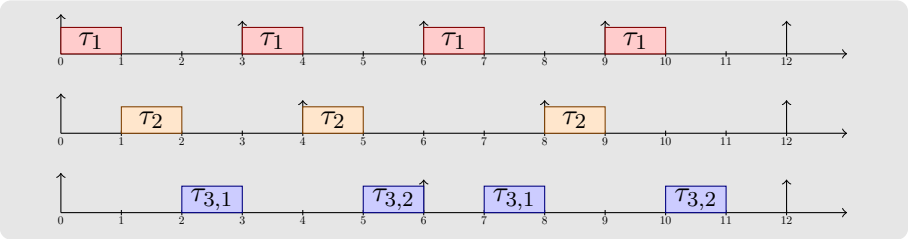


FIGURE 2.2 – Exemple d'ordonnancement selon RM pour trois tâches périodiques : $\tau_1 : C_1 = 1, T_1 = 3$; $\tau_2 : C_2 = 1, T_2 = 4$; $\tau_3 : C_3 = 2, T_3 = 6$

2.3.2 Deadline Monotonic

Rate Monotonic a été proposé avec la contrainte que la date d'échéance relative d'une tâche doit être égale à sa période ($D_i = T_i$). L'algorithme *Deadline Monotonic* (*DM*), proposé et démontré optimal dans sa catégorie par Leung et Whitehead en 1982 [108], relâche cette contrainte ($D_i \leq T_i$). Selon cette politique d'ordonnancement, les priorités sont affectées de façon similaire à Rate Monotonic avec cette différence que *Deadline Monotonic* prend en considération les dates d'échéance relatives. Ainsi, la tâche ayant la plus faible échéance relative obtient la priorité la plus grande. Deadline Monotonic est donc classé dans le groupe des algorithmes préemptifs à priorités fixes pour les tâches périodiques. Dans le cas particulier où les échéances sont égales aux périodes il est équivalent à Rate Monotonic.

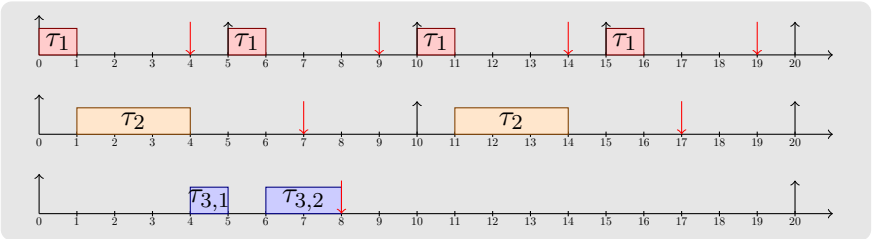


FIGURE 2.3 – Exemple d'ordonnancement selon DM pour trois tâches périodiques : $\tau_1 : C_1 = 1, T_1 = 5, D_1 = 4$; $\tau_2 : C_2 = 3, T_2 = 10, D_2 = 7$; $\tau_3 : C_3 = 3, T_3 = 20, D_3 = 8$

2.3.3 Earliest Deadline First

Earliest Deadline First (*EDF*) est un algorithme à priorité dynamique (fixe au niveau de la requête). Contrairement à Rate Monotonic et Deadline Monotonic, les priorités sont

réévaluées à chaque activation de tâche. Les priorités sont allouées en fonction des dates d'échéance absolues. Les tâches caractérisées par les dates d'échéance les plus proches ont les priorités les plus élevées. En 1973, Liu et Layland [112] ont démontré l'optimalité et formulé une condition nécessaire et suffisante d'ordonnançabilité pour *EDF* appliqué aux tâches périodiques à échéance sur requête ($D_i = T_i$). Le principe de l'algorithme est similaire au cas des tâches apériodiques présenté précédemment dans les travaux de Horn [91] et la preuve de son optimalité est basée sur la règle de Jackson [92, 75]. L'algorithme *EDF* est également optimal pour les tâches à échéance contrainte ($D_i \leq T_i$).

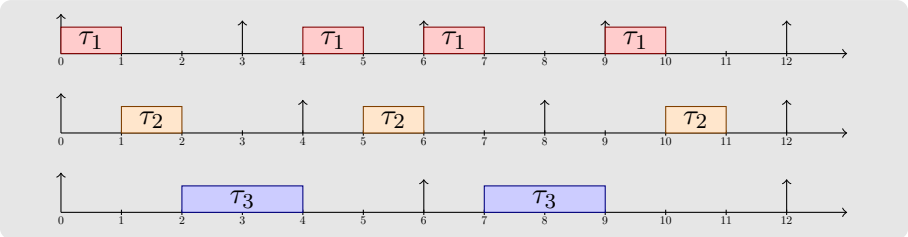


FIGURE 2.4 – Exemple d'ordonnancement selon EDF pour trois tâches périodiques : $\tau_1 : C_1 = 1, T_1 = 3$; $\tau_2 : C_2 = 1, T_2 = 4$; $\tau_3 : C_3 = 2, T_3 = 6$

2.3.4 Least Laxity First

Un autre algorithme à priorité dynamique est celui de *Least Laxity First* (*LLF*) proposé par Mok en 1983 [115]. Contrairement à EDF, la priorité d'une instance de tâche peut varier durant son exécution. Plus précisément, les priorités sont attribuées en fonction de la valeur courante de la laxité donnée par l'équation suivante :

$$L_i(t) = \max(d_i - t - c_i(t), 0) \tag{2.1}$$

où $c_i(t)$ est le temps d'exécution restant nécessaire à l'instant t pour terminer la tâche τ_i. Il est alors évident qu'à l'instant $t + 1$ les laxités de toutes les tâches, sauf celle qui s'est exécutée dans l'intervalle $[t, t + 1]$ et celles dont les valeurs de laxité sont égales à 0, sont réduites. Par conséquent, il est donc possible que dans certains cas la tâche τ_i soit préemptée par une autre tâche qui a été auparavant activée.

2.4 Théorie classique de l'ordonnancement temps réel

En dehors des techniques de simulation, la faisabilité du système peut être étudiée et vérifiée par des tests d'ordonnancabilité. Eu égard aux conclusions qui peuvent en être tirées, deux types de tests peuvent être distingués.

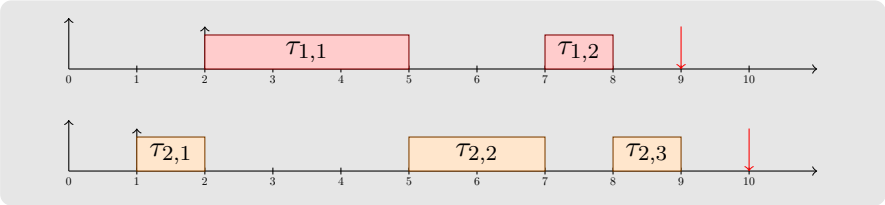


FIGURE 2.5 – Exemple d'ordonnancement selon LLF pour deux tâches apériodiques : $\tau_1 : r_1 = 2, C_1 = 4, D_1 = 9$; $\tau_2 : r_2 = 1, C_2 = 4, D_2 = 10$

Les tests exacts fournissent des conditions nécessaires et suffisantes pour l'ordonnançabilité d'un ensemble des tâches [147].

Les tests approchés sont caractérisés en général par une complexité algorithmique moindre et établissent seulement des conditions suffisantes. A l'inverse des tests exacts, ils ne permettent pas dans tous les cas de déterminer avec certitude si le système est ordonnançable [147].

La mise en place de ces tests fait d'ordinaire appel à l'une des trois techniques exposées ci-dessous. L'analyse de *l'utilisation du processeur* cherche à évaluer le facteur de temps que le processeur passe à exécuter des tâches et à comparer ensuite cette valeur à une valeur seuil caractéristique de la politique d'ordonnancement donnée. Pour un ensemble de n tâches, le facteur d'utilisation du processeur U se définit comme [112] :

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \tag{2.2}$$

Si, pour un ensemble de tâches donné, le facteur d'utilisation est plus grand que 1, il n'existe aucun algorithme qui puisse ordonnancer les tâches de cet ensemble sur un processeur.

Une autre approche, celle de la *demande processeur*, se fonde sur l'analyse de la demande cumulée par les instances de tâches ayant leurs dates d'activation et leurs dates d'échéance situées dans un intervalle donné. S'il n'existe aucun intervalle où la demande est supérieure au temps de processeur disponible, l'ordonnançabilité est prouvée.

Enfin, la troisième méthode, celle de *l'analyse des temps de réponse*, consiste à calculer les pires temps de réponse des tâches. Ces temps sont ensuite comparés aux échéances relatives des tâches et le respect de ces échéances peut alors être vérifié [146, 147].

2.4.1 Priorité fixe

L'instant critique est celui pour lequel le temps de réponse d'une tâche est maximal. Il se produit lorsque toutes les tâches ayant les priorités les plus hautes sont déclenchées simultanément [112].

A partir de cette observation, deux approches pour vérifier l'ordonnançabilité pour les algorithmes à priorité fixe, Rate Monotonic et Deadline Monotonic, ont été proposées : la première, basée sur le taux d'utilisation du processeur, fournit une condition suffisante d'ordonnançabilité et la seconde, basée sur le pire temps de réponse, fournit une condition nécessaire et suffisante d'ordonnançabilité.

Taux d'utilisation du processeur Liu et Layland ont démontré qu'une valeur limite du facteur d'utilisation pour un ensemble de n tâches sous Rate Monotonic est donnée par la formule [112] :

$$U_{lub} = n\left(2^{\frac{1}{n}} - 1\right) \tag{2.3}$$

Sa limite, lorsque le nombre n tend vers l'infini, est égale à :

$$U_{lub} = ln2 \simeq 0,69 \tag{2.4}$$

Ce résultat a été obtenu en lançant un ensemble générique de n tâches à leur instant critique avec leurs pires temps d'exécution choisis de telle façon que cet ensemble utilise pleinement le processeur et que l'augmentation de l'un des temps d'exécution mène au dépassement d'une échéance. Ensuite, en réduisant le facteur d'utilisation de cet ensemble, une valeur minimale en a été trouvée au-dessous de laquelle tout ensemble de n tâches est ordonnançable. Si un ensemble de n tâches est caractérisé par un facteur d'utilisation qui dépasse cette valeur, son ordonnançabilité ne peut pas être déterminée par le test. Lehoczky et al. ont observé que la valeur seuil pour Rate Monotonic égale statistiquement 0.88 [105] ce qui, toutefois, ne peut pas être considéré comme une condition garantissant l'ordonnançabilité. Pour un ensemble de tâches dont toutes les périodes sont harmoniques deux à deux $U_{lub} = 1$ [112, 39].
Un autre test d'ordonnançabilité, ayant la même complexité algorithmique que celui de Liu et Layland et améliorant le taux d'acceptation de celui-ci, a été proposé par Bini et al. [34]. D'après ce test, l'ensemble de n tâches est ordonnançable sous Rate Monotonic si :

$$\prod_{i=1}^{n}\left(\frac{C_i}{T_i} + 1\right) \leq 2 \tag{2.5}$$

Les deux tests mentionnés ci-dessus sont applicables à Deadline Monotonic par le remplacement du terme de la période (T) par le terme de la date d'échéance relative (D) dans les formules appropriées (2.2 et 2.5).

Pire temps de réponse A défaut de réponse positive fournie par les tests basés sur le taux d'utilisation du processeur, la question de l'ordonnançabilité peut être résolue par les tests exacts fondés sur l'estimation du pire temps de réponse.
Lehoczky et al. ont caractérisé un ensemble de tâches ordonnançable sous Rate Monotonic par le théorème suivant [105] :

Test de Lehoczky. *Soient n tâches $\tau_1, \ldots, \tau_n$ mises en ordre par Rate Monotonic et décrites par :*

$$\begin{aligned} W_i(t) &= \sum_{j=1}^{i} C_j \lceil t/T_j \rceil \\ L_i(t) &= W_i(t)/t \\ L_i &= min_{(0<t<T_i)} L_i(t) \\ L &= max_{(1\leq i\leq n)} L_i \end{aligned} \tag{2.6}$$

1. *Une tâche τ_i est ordonnançable sous* Rate Monotonic *si et seulement si $L_i \leq 1$*
2. *Tout ensemble de tâches est ordonnançable sous* Rate Monotonic *si et seulement si $L \leq 1$*

L'analyse de tout l'intervalle $[0, T_i]$ peut se réduire, sans perte de généralité, à un ensemble de points définis comme suit :

$$S_i = \{kT_j | j = 1, \ldots, i; k = 1, \ldots \lfloor T_i/T_j \rfloor\} \tag{2.7}$$

Joseph et Pandya [94] ont déterminé une relation pour le pire temps de réponse d'une tâche qui a été ensuite reprise par Audsley et al. [13] afin de formuler une méthode itérative, avec une complexité pseudo-polynomiale, de son calcul. La méthode est applicable à Rate Monotonic ainsi qu'à Deadline Monotonic. Un ensemble de tâches est ordonnançable si pour toute tâche τ_i de cet ensemble le pire temps de réponse R_i de cette tâche est inférieur ou égal à sa data d'échéance D_i.

$$\begin{cases} R_i^{(0)} = C_i \\ R_i^{(k)} = C_i + \sum\limits_{j:D_j<D_i} \left\lceil \dfrac{R_i^{(k-1)}}{T_j} \right\rceil C_j \end{cases} \tag{2.8}$$

L'équation 2.8 exprime le pire temps de réponse de la tâche τ_i qui peut être observé lorsque celle-ci est déclenchée simultanément avec toutes les tâches plus prioritaires (l'instant critique). En termes de coût de calcul, l'évaluation des interférences des tâches plus prioritaires, représentées par le dernier membre dans l'équation 2.8, est peu efficace (plusieurs itérations peuvent être nécessaires pour évaluer ces interférences). Dans les travaux antérieurs d'Audsley et al. [14, 15] cette interférence, désignée dans ce qui suit par I_i, est évaluée par l'équation ci-dessous.

$$I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{T_j} \right\rceil C_j \tag{2.9}$$

Néanmoins cette expression peut surestimer la valeur d'interférence I_i réelle dans le cas où la dernière instance d'une des tâches plus prioritaires τ_j dont la totalité de temps

d'exécution est prise en compte par cette expression finit son exécution après D_i. Une évaluation plus précise qui limite le temps d'interférence produit par la tâche de priorité plus haute finissant après D_i peut être donnée par :

$$I_i = \sum_{j=1}^{i-1} \left[\left\lfloor \frac{D_i}{T_j} \right\rfloor C_j + min \left(C_j, D_i - \left\lfloor \frac{D_i}{T_j} \right\rfloor T_j \right) \right] \tag{2.10}$$

Cette valeur n'est pas exacte dans tous les cas. Si deux tâches plus prioritaires réclament au même instant le processeur, il est évident que seule une de ces tâches, celle qui possède la priorité plus élevée, l'obtiendra. Cependant, l'égalité 2.10 accumule dans ce cas les temps des instances qui finissent après D_i, comme si elles s'exécutaient parallèlement. Toutefois, pour un ensemble constitué de deux tâches, cette situation ne se produira pas et la formule 2.10 peut être considérée comme une condition nécessaire et suffisante [14]. Dans le cas contraire, celle-ci fournit uniquement une condition suffisante.

2.4.2 Priorité dynamique

Dans les algorithmes à priorité dynamique la priorité d'une tâche évolue au cours du temps. Les interférences qu'une tâche subit pendant son exécution ne proviennent pas toujours, comme c'était le cas dans les algorithmes à priorité fixe, du même ensemble de tâches plus prioritaires. Celui-ci, en fonction des échéances les plus proches, évolue et la hiérarchie des priorités change avec le temps. La prise en compte des interférences n'est plus possible comme dans le cas des systèmes à priorité fixes et la plupart des méthodes destinées à vérifier l'ordonnançabilité du système dont les tâches possèdent des priorités dynamiques sont fondées soit sur le facteur d'utilisation du processeur soit sur la fonction de demande du processeur.

En 1973, Liu et Layland [112] ont démontré l'optimalité et formulé une condition nécessaire et suffisante d'ordonnançabilité pour Earliest Deadline First appliqué aux tâches périodiques à échéance sur requête.

Condition d'ordonnançabilité pour EDF ($D_i = T_i$). *Un ensemble de n tâches à échéance sur requête est ordonnançable sous Earliest Deadline First si et seulement si l'inégalité suivante est satisfaite :*

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.11}$$

Dans le cas plus général, i.e. si les échéances ne sont pas forcement égales aux périodes, la condition d'ordonnançabilité formulée par Liu et Layland [112] est toujours nécessaire mais pas suffisante. Un test adéquat a été proposé par Baruah, Howell et Rosier en 1990 [24] en s'appuyant sur l'analyse de la demande processeur. Avant de l'introduire, il est indispensable de définir *la fonction de demande*, qui quantifie le travail du processeur devant être réalisé dans un intervalle de temps donné.

Définition 2.1. *La fonction de demande $df(t_1, t_2)$ associe à chaque intervalle $[t_1, t_2]$ tel que $t_1 \leq t_2$ la durée d'exécution cumulée des instances de tâches dont la date d'activation et d'échéance sont dans cet intervalle.*

$$df(t_1, t_2) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \eta_i(t_1, t_2) \cdot C_i \tag{2.12}$$

où $\eta_i(t_1, t_2)$ est le nombre d'exécutions de la tâche τ_i terminées dans l'intervalle $[t_1, t_2]$.

Le nombre des exécutions terminées d'une tâche dans l'intervalle $[t_1, t_2]$ s'exprime par la différence entre le nombre de ses échéances antérieures à t_2 et le nombre des requêtes antérieures à t_1. Par conséquent, il peut être défini pour une tâche périodique comme suit :

$$\eta_i(t_1, t_2) \stackrel{\text{def}}{=} \max\left(0, \left\lfloor \frac{t_2 + T_i - D_i}{T_i} \right\rfloor - \left\lceil \frac{t_1}{T_i} \right\rceil\right) = \max\left(0, \left\lfloor \frac{t_2 - D_i}{T_i} \right\rfloor - \left\lceil \frac{t_1}{T_i} \right\rceil + 1\right) \tag{2.13}$$

Intuitivement, tous les calculs qui doivent être réalisés entre les instants t_1 et t_2 ne peuvent dépasser la capacité d'un processeur dans cet intervalle :

$$df(t_1, t_2) \leq t_2 - t_1 \tag{2.14}$$

S'il existe un intervalle tel que le temps d'exécution requis par les tâches excède le temps disponible d'un processeur, la configuration d'ordonnancement n'est pas faisable. Pour que le test puisse être praticable, la longueur de l'intervalle doit être bornée. Leung et Merrill ont démontré que chaque séquence d'ordonnancement produite par une politique déterministe préemptive se répète cycliquement, à partir d'un certain instant t, en prenant les mêmes décisions à l'instant t qu'à l'instant $t + kH$, où H désigne *l'hyperpériode*, i.e. le plus petit commun multiple des périodes des tâches [107]. Ils ont observé qu'un ensemble de tâches est ordonnançable sur un processeur si toutes les échéances dans l'intervalle $[0, \max\{\phi_1, \ldots, \phi_n\} + 2H]$ sont respectées. Cette borne a été ensuite améliorée par Choquet-Geniet, Grolleau et Cottet en démontrant que le schéma d'ordonnancement est cyclique, de période H, à partir de son dernier temps creux (l'instant oisif) acyclique [51, 76, 77].

Condition d'ordonnançabilité pour EDF ($D_i \leq T_i$). *Un ensemble de n tâches est ordonnançable sur un processeur sous Earliest Deadline First si et seulement si :*

1. $\displaystyle\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1$, *et*
2. $df(t_1, t_2) \leq t_2 - t_1$ *pour* t_1 *et* t_2 *tels que :* $0 \leq t_1 < t_2 \leq \max\{\phi_1, \ldots, \phi_n\} + 2H$

Démonstration. Les deux conditions sont évidement nécessaires. On prouve qu'elles sont suffisantes. On procède par l'absurde. On suppose que les conditions 1 et 2 sont satisfaites. On suppose également que l'ordonnancement n'est pas faisable. Conformément aux observations faites par Leung et Merrill [107], dans l'intervalle $[0, \max\{\phi_1, \ldots, \phi_n\} + 2H]$ il doit y

avoir une tâche τ dont l'échéance est dépassée. Soit t_2 cette échéance. La tâche τ ne peut être empêchée de s'exécuter que par des instances de tâches dont les dates d'échéance sont inférieures ou égales à t_2. Soit $t_1 - 1$ le dernier instant inférieur à t_2 tel qu'aucune instance de tâche dont l'échéance est antérieure ou égale à t_2 ne s'exécute à $t_1 - 1$. Il existe donc une tâche dont la date d'échéance est antérieure ou égale à t_2 et qui est activée exactement à l'instant t_1. Earliest Deadline First est un algorithme conservatif donc le processeur est occupé dans l'intervalle $[t_1, t_2]$. Cela implique, puisque une tâche dépasse son échéance à t_2, que $df(t_1, t_2) > t_2 - t_1$ et contredit l'hypothèse de départ. □

Le problème de l'ordonnançabilité se résout en temps exponentiel. Sa complexité est cependant réduite dans le cas particulier de tâches synchrones. La faisabilité d'un système synchrone implique de plus la faisabilité de tous les systèmes équivalents asynchrones [148, 74]. De même, si les périodes de toutes les tâches sont des entiers premiers entre eux, le test pour un système synchrone donne un résultat exact pour tous les systèmes asynchrones qui peuvent en être dérivés [148]. Leung et Merrill [107] ont prouvé que, dans le cas synchrone, il est suffisant de limiter les intervalles de test à l'hyperpériode. Baruah et al. [24] ont fixé une borne au test de faisabilité par la condition suivante :

$$t_2 < \frac{\sum_{i=1}^{n} C_i \left(1 - \frac{D_i}{T_i}\right)}{1 - U} \leq \frac{U}{1 - U} \max_i \{T_i - D_i\} \tag{2.15}$$

Ils ont aussi observé que dans le cas synchrone $\eta_i(0, t_2 - t_1) \geq \eta_i(t_1, t_2)$, ce qui permet de considérer dans ce cas seulement les intervalles qui commencent à l'instant 0. Cette observation permet aussi de constater que la durée cumulée d'exécution dans l'intervalle $[0, t_2 - t_1]$, pour un ensemble de tâches à départ simultané, est toujours supérieure ou égale à celle d'un ensemble équivalent de tâches asynchrones dans l'intervalle $[t_1, t_2]$. Cela plaide pour ne considérer, dans le cas synchrone, que les intervalles à partir de 0 (le pire cas).

Définition 2.2. *Soit $\Delta > 0$. La fonction $dbf(\Delta)$ dénote la plus grande durée d'exécution cumulée des instances de tâche dont la date d'activation et l'échéance sont dans n'importe quel intervalle de durée Δ [25, 27].*

$$dbf(\Delta) = \max_{t_1} df(t_1, t_1 + \Delta) \tag{2.16}$$

Compte tenu de cette définition, la condition d'ordonnançabilité prend la forme suivante :

$$\forall \Delta : dbf(\Delta) \leq \Delta \tag{2.17}$$

Liu et Layland [112] ont observé que le premier dépassement d'une échéance ne survient jamais après une période où le processeur est, même momentanément, inactif. Il est donc possible de restreindre l'analyse des intervalles jusqu'au premier instant d'inactivité du processeur [160, 148, 38]. La demande totale de calcul $W(t)$ à un instant t est donnée par :

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{T_i} \right\rceil C_i \tag{2.18}$$

Le processeur est inoccupé à l'instant t si toute sa demande est déjà exécutée [160, 38], c'est-à-dire $W(t) = t$. La longueur L de l'intervalle pendant lequel le processeur est continuellement occupé peut être obtenue par la formule itérative présentée ci-dessous. Elle est calculée jusqu'à ce que $L^{(m)} = L^{(m+1)}$:

$$\begin{cases} L^{(0)} = \sum_{i=1}^{n} C_i \\ L^{(m+1)} = W(L^{(m)}) \end{cases} \tag{2.19}$$

Cet intervalle est limité par la borne [148] :

$$L < \frac{\sum_{i=1}^{n} C_i}{1 - U} \tag{2.20}$$

Ces bornes sont calculées pour un modèle de tâches purement périodiques. Dans le cas de processus dont les enchaînements ne peuvent pas être décrits par une relation périodique et révèlent ainsi une nature plus complexe, d'autres modèles doivent bien évidemment être utilisés.

Le modèle *Multiframe Tasks* [116, 27] décompose une tâche en plusieurs sous-tâche, chacune définie par ses propres paramètres d'exécution et ses propres contraintes. Les activations se succèdent à intervalles fixes mais pas nécessairement réguliers. Le modèle des *transactions* [127] étend le *Multiframe* (les priorités des sous-tâches peuvent être différentes, les sous-tâches peuvent s'entrelacer, la gigue est prise en compte [75]).

Le modèle *Recurring Branching Task* [25] ajoute la possibilité de prendre en compte les différentes séquences d'exécution d'une tâche. Cette dernière est représentée sous la forme d'un arbre binaire dont les nœuds sont constitués par les sous-tâches et dont les arêtes reflètent le flot d'exécution de la tâche. Formellement, un nœud est caractérisé par le pire temps d'exécution e_i de la sous-tâche correspondante et par sa date d'échéance relative d_i. Une arête reliant deux sous-tâches, S_i et S_j, est étiquetée par l'intervalle p_{ij} qui sépare leurs activations consécutives. La période d'une tâche τ_i est toujours constante quel que soit le parcours emprunté depuis la racine jusqu'aux feuilles. Cette période est notée $T(\tau_i)$ et le temps maximal de processeur demandé pendant cet intervalle noté $E(\tau_i)$. Avec cette définition des tâches, la borne de faisabilité peut être obtenue comme suit. Dans l'intervalle Δ la tâche τ_i demande au maximum $dbf(\tau_i, \Delta)$ des ressources du processeur. Elle y exécute un nombre de périodes entières, précédé et suivi par une exécution partielle. Cette exécution partielle s'étend, pour la partie qui précède, depuis un nœud jusqu'à la racine, et pour celle qui succède, depuis la racine jusqu'à un nœud de la tâche. Les deux exécutions partielles peuvent être synthétisées en un intervalle pendant lequel la sous-tâche de la racine est exécutée exactement une fois. Dans les limites de cet intervalle, $\rho_{max}(\tau_i)$ décrit le rapport le plus élevé entre la demande du temps de processeur et l'intervalle dans lequel cette demande a été faite. Pareillement, pendant l'exécution complète d'une période, ce rapport est donné par $\rho_{ave}(\tau_i) = E(\tau_i)/T(\tau_i)$. Cela permet de borner la fonction $dbf(\tau_i, \Delta)$ par la relation suivante : $dbf(\tau_i, \Delta) \leq 2T(\tau_i)\rho_{max}(\tau_i) + \Delta\rho_{ave}(\tau_i)$. D'après Baruah, un système

de n tâches est infaisable s'il existe un intervalle Δ tel que $\Delta < \sum_{i=1}^{n} dbf(\tau_i, \Delta)$. Cette condition peut être réécrite en bornant la fonction $dbf(\tau_i, \Delta)$ par la relation précédemment obtenue et constitue une borne pour la longueur maximale des intervalles à vérifier :

$$\Delta < \frac{2\sum_{i=1}^{n}(T(\tau_i)\rho_{max}(\tau_i))}{1-\sum_{i=1}^{n}\rho_{ave}(\tau_i)} \tag{2.21}$$

Plusieurs techniques ont été développées dans la littérature en vue de réduire la complexité des tests. Des approximations de la fonction dbf [44, 5, 6] permettent, en introduisant un certain taux d'erreur, d'estimer sa valeur plus efficacement. L'idée maîtresse de ces approches consiste à évaluer précisément la demande d'un nombre donné de premières instances de tâches et d'estimer le reste à l'aide d'une fonction linéaire.

D'autre part, dans le cas de tâches asynchrones, certaines relations entre les instants de démarrage des tâches peuvent être exploitées pour atténuer le pessimisme de l'approche synchrone sans atteindre cependant la complexité exponentielle de la solution exacte. Pellizzoni et Lipari [132, 133] ont déterminé l'intervalle minimal Δ_{ij} qui peut séparer deux activations consécutives des tâches τ_i et τ_j. Cet intervalle s'exprime comme la différence entre les instants d'activation $r_{j,l}$ et $r_{i,m}$ en supposant que $r_{i,m}$ ait lieu avant $r_{j,l}$: $\Delta_{ij} = r_{j,l} - r_{i,m} = \phi_j - \phi_i + l \cdot T_j - m \cdot T_i$ et $\Delta_{ij} \geq 0$. En posant $l \cdot T_j - m \cdot T_i = K \cdot pgcd(T_i, T_j)$, pour $K \in \mathbb{Z}$, on obtient : $\Delta_{ij} = \phi_j - \phi_i + K \cdot pgcd(T_i, T_j)$. La valeur de Δ_{ij}, qui doit être minimisée, croît avec K. En tenant compte du fait que $\Delta_{ij} \geq 0$ et en regroupant les termes, la variable $K \in \mathbb{Z}$ est bornée inférieurement par la relation suivante : $K \geq \lceil \frac{\phi_i - \phi_j}{pgcd(T_i,T_j)} \rceil$. L'intervalle minimal Δ_{ij} est donc égal à :

$$\Delta_{ij} = \phi_j - \phi_i + \left\lceil \frac{\phi_i - \phi_j}{pgcd(T_i, T_j)} \right\rceil \cdot pgcd(T_i, T_j) \tag{2.22}$$

Le test consiste, d'abord, à transformer l'ensemble de tâches initial en un nouvel ensemble de tâches où les phases des tâches sont remplacées par les intervalles minimaux qui séparent leurs activations de celle de la première tâche de cet ensemble. Ensuite, les charges du processeur dans tous les intervalles, s'étendant à partir de l'instant 0 jusqu'à tout instant avant l'instant oisif, sont analysées. Les auteurs reconnaissent que ce test fournit seulement une condition suffisante et, dans certains cas, sous-estime les intervalles minimaux qui séparent les dates d'activation. L'exemple suivant illustre ce défaut [132, 133].

Exemple 2.1. *On considère trois tâches τ_1,τ_2 et τ_3 telles que $\tau_1(\phi_1 = 0, C_1 = 1, D_1 = 2, T_1 = 5)$, $\tau_2(\phi_2 = 1, C_2 = 1, D_2 = 2, T_2 = 4)$, $\tau_3(\phi_3 = 2, C_3 = 1, D_3 = 2, T_3 = 6)$. Grâce à l'équation 2.22, les intervalles minimaux entre les dates d'activation des tâches sont obtenus comme suit : $\Delta_{12} = \Delta_{13} = 0$. Le test revient donc à celui d'un système synchrone et échoue alors à prouver l'ordonnançabilité de l'ensemble de tâches considérées comme synchrones. Or, la distance minimale Δ_{23} entre les activations des tâches τ_2 et τ_3 est, toujours selon 2.22, égale à 1. Donc contrairement à l'hypothèse formulée, les trois*

tâches ne seront en fait jamais activées simultanément et la conclusion du test est sans doute trop pessimiste.

L'exemple 2.1 montre que c'est en considérant les relations entre toutes les tâches que le schéma d'exécution réel peut être reproduit. Pellizzoni et Lipari étudient ces relations dans leurs travaux [132, 133]. Ils proposent un nouveau test qui retrace les schémas d'exécution exacts de certaines tâches et traitent d'autres tâches avec les distances obtenues à partir de l'équation 2.22 pour ne pas augmenter excessivement la complexité du test.
Tous les intervalles entre activations les plus récentes de deux tâches doivent être examinés. Si $r_{i,l}$ est une date d'activation de la tâche τ_i et $r_{j,p}$ est la première date d'activation de la tâche τ_j depuis celle de la τ_i, les distances possibles entre elles appartiennent à l'ensemble suivant :

$$\left\{ \Delta_{ij}(k) \;\mid\; \forall 0 \leq k < \frac{T_j}{pgcd(T_i,T_j)} \right\}, \quad o\grave{u} \quad \Delta_{ij}(k) = \left\lceil \frac{\phi_i + k \cdot T_i - \phi_j}{T_j} \right\rceil T_j - (\phi_i + k \cdot T_i - \phi_j) \tag{2.23}$$

La relation ci-dessus fixe une distance entre les activations de deux tâches. Maintenant, si la tâche τ_2 démarre $\Delta_{12}(k_1)$ après le démarrage de τ_1, il est essentiel de trouver quelles valeurs de distances sont admissibles pour d'autres tâches $\tau_3, \ldots, \tau_m$. La configuration dans laquelle les démarrages des tâches τ_1 et τ_2 sont séparés par une distance de $\Delta_{12}(k_1)$, survient périodiquement à des intervalles $ppcm(T_1, T_2)$. Un instant t auquel la tâche τ_1 démarre dans cette configuration est donné par $t = \phi_1 + k_1 \cdot T_1 + n \cdot ppcm(T_1, T_2)$. Il est alors possible de considérer cette configuration comme une super-tâche $\tau_{1\otimes 2}$ dont la période serait $ppcm(T_1, T_2)$ et la phase $\phi_1 + k_1 T_1$. Les distances valides entre les dates d'activation des tâches τ_3 et τ_1, peuvent être obtenues comme celles séparant les dates d'activation de τ_3 et $\tau_{1\otimes 2}$.

Exemple 2.2. *Le tableau ci-dessous résume les distances entre les plus récentes dates d'activation pour les tâches de l'exemple précédent. Ces distances ont été calculées à partir de la formule 2.23. On note $\Delta_{(12)3}(k_2)$ l'intervalle séparant l'activation de τ_3 de celle de $\tau_{1\otimes 2}$.*

Les valeurs de k_1 et k_2 sont calculées selon les formules suivantes :

$$k_1 \in \left\{ 0, \ldots, \frac{T_2}{pgcd(T_1,T_2)} \right\} = \{0,1,2,3\}, \; k_2 \in \left\{ 0, \ldots, \frac{T_3}{pgcd\left(ppcm\left(T_1,T_2\right),T_3\right)} \right\} = \{0,1,2\}$$

Les distances entre les dates d'activation de la tâche τ_1 et les dates d'activations de la tâche τ_2, $\Delta_{12}(k_1)$, sont données comme suit :

$$\Delta_{12}(k_1) = \left\lceil \frac{\phi_1 + k_1 \cdot T_1 - \phi_2}{T_2} \right\rceil T_2 - (\phi_1 + k_1 \cdot T_1 - \phi_2)$$

k_1	k_2	$\Delta_{12}(k_1)$	$\Delta_{(12)3}(k_2)$	k_1	k_2	$\Delta_{12}(k_1)$	$\Delta_{(12)3}(k_2)$
0	0	1	2	2	0	3	4
0	1	1	0	2	1	3	2
0	2	1	4	2	2	3	0
1	0	0	3	3	0	2	5
1	1	0	1	3	1	2	3
1	2	0	5	3	2	2	1

TABLE 2.1 – Distances entre dates d'activation des tâches pour l'Exemple 2.1

et celles entre les dates d'activation de la tâche τ_1 *et les dates d'activations de la tâche* τ_3, $\Delta_{(12)3}(k_2)$, *comme suit :*

$$\Delta_{(12)3}(k_2) = \left\lceil \frac{\phi_1 + k_1 \cdot T_1 + k_2 \cdot ppcm(T_1, T_2) - \phi_3}{T_3} \right\rceil T_3 - (\phi_1 + k_1 \cdot T_1 + k_2 \cdot ppcm(T_1, T_2) - \phi_3)$$

D'autres tâches peuvent ensuite être intégrées dans la configuration de tâches déjà fixée en suivant le même raisonnement. Jelkmann [93] a combiné le test de Pellizzoni et Lipari avec des méthodes de calcul approximatif de la fonction de demande afin de réduire la complexité du test.

2.5 Analyses et protocoles pour le changement de mode

Pour les raisons déjà discutées dans la première partie de ce manuscrit (voir section 1.2), on attend du système qu'il puisse changer son mode de fonctionnement en fonction des évolutions observées dans son environnement. Le changement de mode, du point de vue de l'ordonnancement, s'opère sur un ensemble de tâches. Il consiste à supprimer ou à modifier les paramètres de certaines d'entre elles et à en introduire de nouvelles. Les différents protocoles de changement de mode étudiés dans cette section définissent la manière dont les tâches de l'ancien mode mettent un terme à leurs exécutions, dont celles du nouveau mode sont introduites, et dont celles qui sont communes aux deux modes traversent la phase transitoire.

La Figure 2.6 résume les principaux types de tâches rencontrés lors d'un changement de mode. L'exécution d'une tâche de l'ancien mode peut être immédiatement abandonnée à l'instant du changement de mode ($\tau_{abandonnée}$) ou bien, si le résultat produit par cette tâche est vital pour le bon fonctionnement du système, il est permis que l'exécution de sa dernière instance s'achève normalement ($\tau_{terminée}$). L'activation d'une tâche du nouveau mode ($\tau_{nouvelle}$) peut être retardée un certain laps de temps afin de réduire la charge lorsque les tâches de l'ancien mode requièrent du temps pour terminer leurs dernières instances. Les tâches présentes dans les modes ancien et nouveau peuvent continuer leur exécution avec les mêmes paramètres ($\tau_{inchangée}$) ou avec un ou plusieurs paramètres modifiés ($\tau_{changée}$).

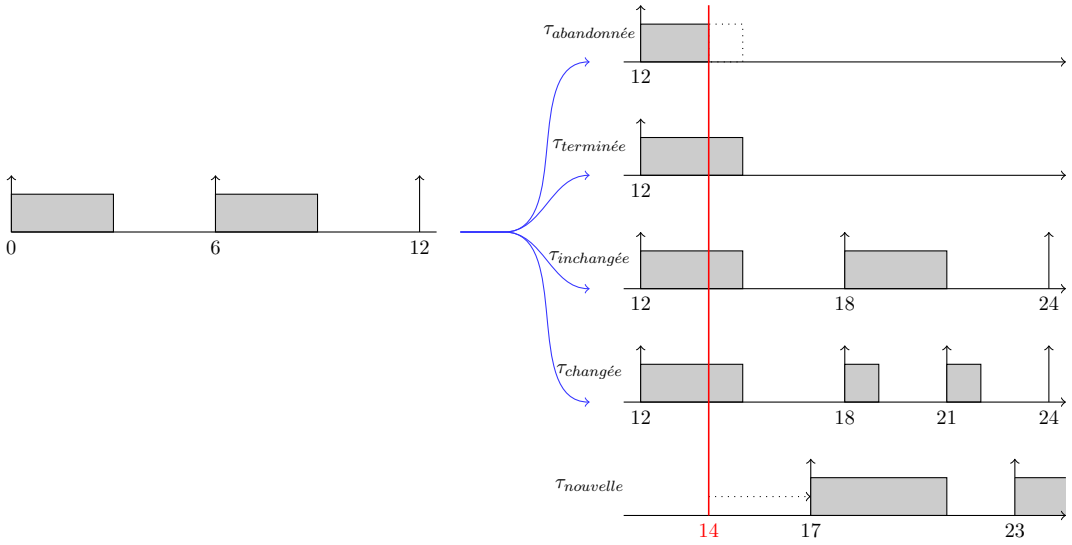


FIGURE 2.6 – Classification des tâches pendant le changement de mode.

Les nouvelles versions des tâches inchangées et changées sont déclenchées immédiatement à la fin de la période des dernières instances activées dans l'ancien mode ou après un certain délai suivant cet instant. Si les schémas d'exécution des tâches inchangées ne sont pas affectés par le changement de mode on dit que la *périodicité* des tâches est préservée.

Les protocoles de changement de mode *synchrones* retardent l'instant du lancement des tâches dans le nouveau mode jusqu'à ce que toutes les tâches de l'ancien mode soient terminées. Au contraire, les protocoles *asynchrones* permettent que les tâches de ces deux modes s'exécutent simultanément pendant la phase transitoire.

2.5.1 Protocoles synchrones

Dans les protocoles synchrones les tâches du nouveau mode ne sont activées que lorsque toutes les tâches de l'ancien mode sont achevées. Cette séparation entre tâches des deux modes successifs fait de l'ordonnançabilité de chacun des modes du système une condition nécessaire et suffisante de l'ordonnançabilité du système entier. Outre qu'ils simplifient l'analyse d'ordonnançabilité, les protocoles de cette classe sont faciles à implanter et assurent la cohérence entre tâches des différents modes. Le délai maximal entre la demande de changement de mode et le lancement du nouveau mode constitue le principal paramètre caractérisant ces protocoles. Les protocoles synchrones, ne pouvant rivaliser en terme de réactivité avec les protocoles asynchrones, ils étaient un peu moins abordés dans la littérature. Pourtant, avec la montée de l'intérêt suscité par les solutions multiprocesseurs ils ont joué un rôle majeur, grâce à leur simplicité et à leur universalité, dans la définition de

nouveaux protocoles, dont en particulier ceux décrits dans la section 2.5.3 de ce manuscrit.

2.5.1.1 Protocole des instants oisifs

Le principe de ce protocole proposé en 1996 par Tindell et Alonso [171] consiste à repousser l'activation des tâches du nouveau mode au prochain instant oisif, c'est-à-dire, au moment où le processeur aura achevé d'exécuter toutes les tâches courantes et sera inactif. Malgré les avantages apportés par le synchronisme, ce protocole présente une relativement faible réactivité. Dans le pire cas, l'ensemble des tâches ordonnées sous *EDF* occupe 100% de temps du processeur et la demande de changement de mode est intervenue juste après l'instant critique ne pourra être effectuée qu'à l'instant critique suivant.

Exemple 2.3. *Soit un système où deux modes, $mode_I$ et $mode_{II}$, s'exécutent. Le premier mode est composé de deux tâches τ_1 et τ_2 telles que : $\tau_1(C_1 = 10, T_1 = 20)$ et $\tau_2(C_2 = 10, T_2 = 30)$. Dans le $mode_{II}$ la tâche τ_1 est remplacée par une tâche $\tau_{1'}$ telle que $\tau_{1'}(C_{1'} = 20, T_{1'} = 40)$ et la tâche τ_2 par une tâche $\tau_{2'}$ donnée par $\tau_{2'}(C_{2'} = 10, T_{2'} = 20)$. Toutes les tâches, dans les deux modes, sont ordonnancées sous* EDF*. Le diagramme gauche de la Figure 2.7 illustre l'exécution normale du $mode_I$ ininterrompu par une demande de changement de mode. L'axe situé en bas présente les instants d'inactivité du processeur. Par exemple, chaque demande de changement de mode arrivant dans l'intervalle $[40, 50]$ sera réalisée exactement à l'instant 50. Le diagramme de droite dans la Figure 2.7 illustre cette situation.*

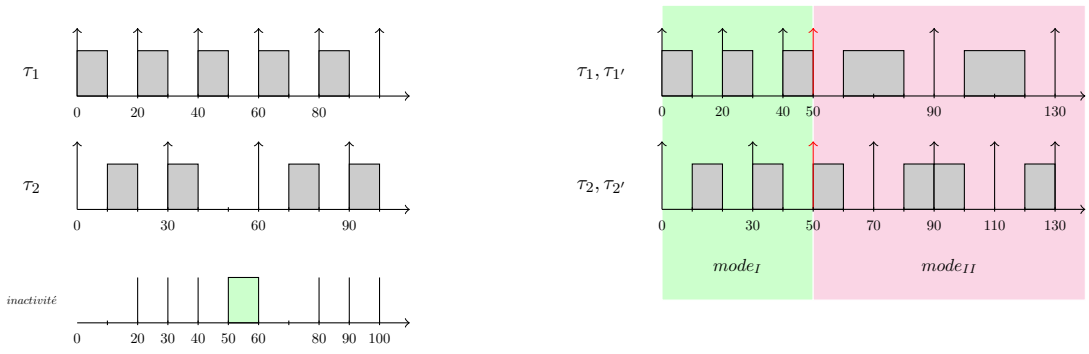


FIGURE 2.7 – A gauche : L'exécution du $mode_I$ et le temps oisif du processeur. A droite : Le changement du $mode_I$ vers le $mode_{II}$ suite à une demande faite lors de l'intervalle $[40, 50]$.

2.5.1.2 Protocole du délai minimal (Minimum Single Offset)

Lors d'un changement de mode, afin de réduire le temps nécessaire au lancement des tâches du nouveau mode, Real a proposé [144] deux versions, avec et sans périodicité,

d'un protocole dénommé *Minimum Single Offset (MSO)*. Contrairement à ce qu'autorise le protocole décrit précédemment, les nouvelles instances de tâches de l'ancien mode ne sont plus activées dès que la demande de changement de mode est émise. Bien entendu, les instances déjà activées s'achèvent normalement. Ainsi, la version du protocole où la périodicité n'est pas exigée est caractérisée par le délai après lequel les tâches du nouveau mode peuvent être introduites. Si *terminées* désigne les tâches devant se terminer dans l'ancien mode et *changées* désigne les tâches dont les paramètres sont modifiés dans le nouveau mode, le délai Y après lequel les tâches du nouveau mode commencent leur exécution est donné par la formule suivante :

$$Y = \sum_{j \in anciennes \cap changées} C_j \tag{2.24}$$

Cela correspond au pire cas, celui où la demande de changement de mode est survenue à l'instant critique quand toutes les tâches commencent leur exécution.

En vue d'obtenir le délai relatif à la version du protocole qui conserve la périodicité, il est nécessaire de tenir compte de toutes les instances des tâches inchangées dont les activations vont devoir apparaître toujours selon le même modèle et sans aucune perturbation ni retard lié au changement de mode en cours :

$$Y = \sum_{j \in anciennes} C_j + \sum_{j \in inchangées} \left\lceil \frac{Y}{T_j} \right\rceil C_j \tag{2.25}$$

Il se peut que certaines tâches inchangées échangent des données, essentielles à leur fonctionnement, avec les tâches de l'ancien mode, arrêtées durant la phase transitoire, et leurs successeures dans le mode suivant qui, ne sont actives qu'après cet intervalle Y. La cohérence du système peut être alors mise en danger.

Exemple 2.4. *Soient trois tâches : τ_1, τ_2 et τ_3. Les tâches peuvent s'exécuter dans un des deux modes : $mode_I$ ou $mode_{II}$. Les paramètres des tâches, selon le mode d'exécution, sont présentés dans le Tableau 2.4*

tâche	$mode_I$		$mode_{II}$	
	C	T	C	T
τ_1	*1*	*3*	*1*	*3*
τ_2	*2*	*4*	*1*	*4*
τ_3	*1*	*6*	*2*	*6*

TABLE 2.2 – Tâches de l'Exemple 2.4

Dans la version du protocole MSO *sans périodicité, le délai Y après lequel les tâches sont activées dans le $mode_{II}$ est calculé selon l'Equation 2.24 :*

$$Y = 1 + 2 + 1 = 4$$

Les tâches τ_1, τ_2 et τ_3 sont toutes considérées comme des tâches changées. La Figure 2.8 illustre un changement de mode opéré selon le protocole MSO *(la version sans périodicité) pour l'ensemble de tâches dont il est question.*

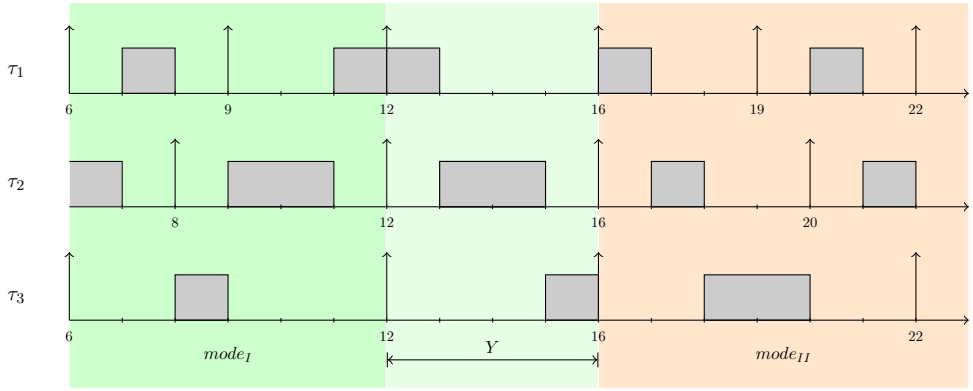


FIGURE 2.8 – Minimum Single Offset sans périodicité

Dans la version du protocole MSO *avec périodicité, le délai Y après lequel les tâches sont activées dans le $mode_{II}$ est calculé selon l'Equation 2.25. Le calcul est réalisé en appliquant une méthode itérative. La tâche τ_1 est considérée comme une tâche inchangée tandis que les autres tâches, τ_2 et τ_3, comme des tâches anciennes.*

$$Y^{(0)} = 2 + 1 + \left\lceil \frac{0}{3} \right\rceil 1 = 4$$
$$Y^{(1)} = 2 + 1 + \left\lceil \frac{4}{3} \right\rceil 1 = 5$$
$$Y^{(2)} = 2 + 1 + \left\lceil \frac{5}{3} \right\rceil 1 = 5$$

La Figure 2.9 illustre un changement de mode opéré selon la version du protocole MSO *conservant la périodicité pour l'ensemble de tâches données dans le Tableau 2.4.*

2.5.2 Protocoles asynchrones

Puisque dans les protocoles asynchrones les tâches du nouveau mode sont déclenchées pendant que celles de l'ancien mode sont encore en train de s'exécuter, la réactivité du changement de mode est meilleure que dans le cas des protocoles synchrones. La superposition momentanée de deux modes dans la phase transitoire produit temporairement une charge de processeur plus élevée. Il est donc primordial, afin d'assurer l'ordonnançabilité du système, d'évaluer correctement la charge supplémentaire engendrée par la transition. Des tests adéquats sont proposés pour différentes politiques d'ordonnancement et règles de changement de mode.

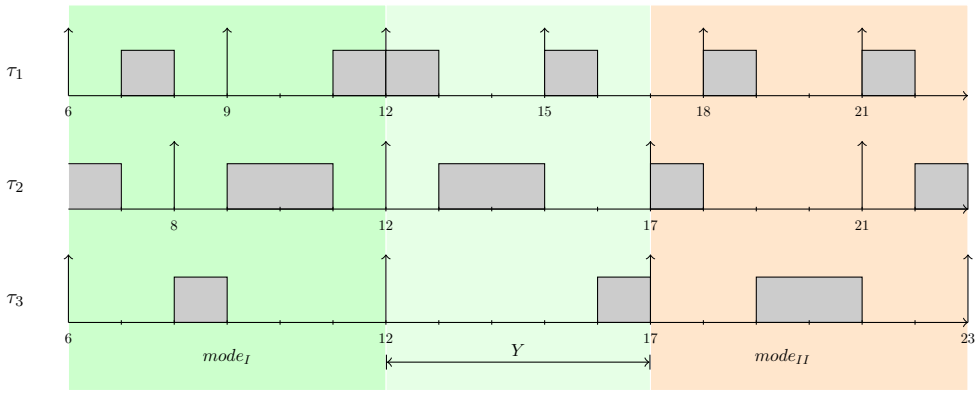


FIGURE 2.9 – Minimum Single Offset avec périodicité

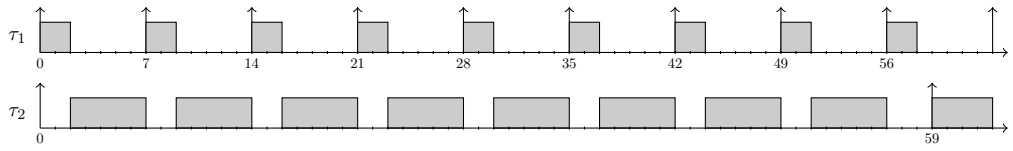
2.5.2.1 Priorité fixe

L'analyse d'ordonnançabilité pour les politiques à priorité fixe repose en général, comme expliqué en 2.4.1, sur deux calculs : celui du taux d'utilisation et celui du pire temps de réponse. Le calcul du taux d'utilisation a d'abord été utilisé pour vérifier la faisabilité d'un ensemble de tâches dont le changement de mode obéit au premier protocole décrit dans cette section (protocole de Sha, Rajkumar, Lehoczky et Ramamritham). Puis, il s'est avéré que le calcul du taux d'utilisation ne permet pas, dans la phase transitoire dans laquelle la régularité des activations des tâches est temporairement perturbée, de fournir une estimation fiable de la charge effective du système durant cette période. Les protocoles suivants sont donc plutôt basés sur le calcul du pire temps de réponse qui offre le moyen d'évaluer, sur un intervalle de temps s'étendant sur deux modes différents, les interférences que subit une tâche moins prioritaire imputable à l'exécution des tâches plus prioritaires des modes différents. Ces protocoles se différencient entre eux par les comportements adoptés par les tâches au regard du changement de mode. Leur analyse est construite, dans tous les cas, sur la base de fenêtres temporelles qui permettent d'observer, à différents moments de passage d'un mode à un autre, quelles tâches, à partir de quand et jusqu'à quand, sont exécutées. Un groupe de tâches adopte un comportement bien précis pendant la transition en fonction des règles définies par chacun des protocoles. L'analyse doit situer, au sein de cette fenêtre d'observation, l'exécution de chaque tâche plus prioritaire s'exécutant en préemptant la tâche dont le pire temps de réponse est recherché. Une fois que ce pire temps est déterminé, il est comparé à la date d'échéance et le respect de cette dernière peut, ou non, être constaté.

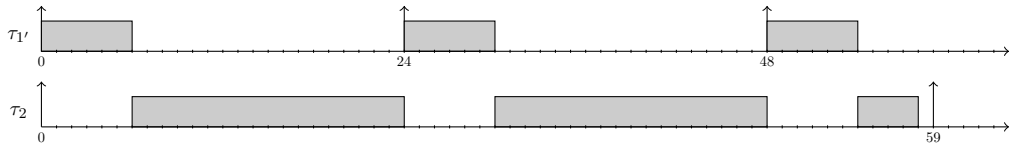
Protocole de Sha, Rajkumar, Lehoczky et Ramamritham Lehoczky a formulé [105] un test exact pour vérifier l'ordonnancement d'un ensemble de tâches exécutées sous *Rate*

Monotonic. En se basant sur ces travaux Sha et al. ont défini [151] un protocole déterminant comment les tâches peuvent être enlevées et insérées dans cet ensemble. Ce protocole permet qu'une nouvelle tâche soit intégrée à un ensemble de tâches à la condition que l'ensemble nouvellement créé soit également ordonnançable. Les ressources temporelles libérées suite à la suppression d'une tâche ne sont utilisables par les autres tâches qu'après la fin de la période de la tâche supprimée. Bien que ce règlement semble bien-fondé, Tindell et al. ont présenté [172] un contre-exemple qui relève l'insuffisance de ce protocole sous certaines conditions.

Contre-exemple de Tindell. *Soit un système pouvant se trouver dans un des deux modes A ou B. Dans le mode A, s'exécutent les deux tâches* τ_1 *et* τ_2 *telles que :* $\tau_1 : C_1 = 2, T_1 = D_1 = 7$ *et* $\tau_2 : C_2 = 40, T_2 = D_2 = 59$. *Dans le mode B, la tâche* τ_1 *est remplacée par une tâche* $\tau_{1'}$ *:* $\tau_{1'} : C_{1'} = 6, T_{1'} = D_{1'} = 24$. *D'après l'équation 2.2 les taux d'utilisation des modes A et B s'élèvent respectivement à 96% et 92%. Malgré ces taux d'utilisation, dépassant la valeur seuil de l'algorithme, le test de Lehoczky (voir équations 2.6) montre que ces deux ensemble sont ordonnançables. Ci-dessous les diagrammes illustrant les exécutions du mode A et du mode B sont présentés.*



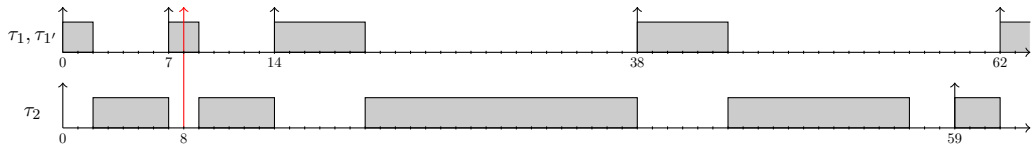
(a) Mode A



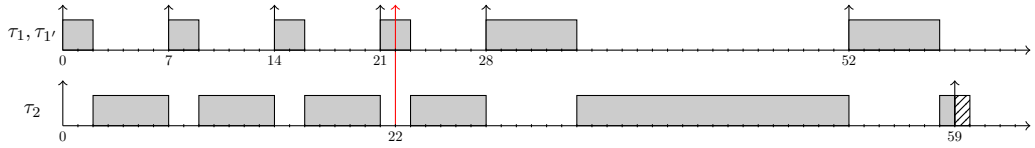
(b) Mode B

FIGURE 2.10 – Exécution des mode A et B sans changement de mode

La transition du mode A vers B à l'instant 8 est tout d'abord analysée. Conformément au protocole, la tâche $\tau_{1'}$ *est déclenchée pour la première fois après la fin de la période de la tâche qui la précède à l'instant 14. La Figure 2.11(a) montre l'ordonnançabilité de cette transformation.*
Si le changement de mode se produit à l'instant 22, la date d'échéance de la tâche τ_2 *ne sera pas respectée comme le montre la Figure 2.11(b). Le protocole ne garantit donc pas toujours que le changement de mode soit réalisable.*



(a) Changement de mode à l'instant 8



(b) Changement de mode à l'instant 22

FIGURE 2.11 – Changement de mode A à mode B

Protocole de Tindell, Burns et Wellings Ayant démontré l'insuffisance de l'approche présentée ci-dessus, Tindell, Burns et Wellings ont proposé [172] leur propre protocole qui permet à un ensemble de tâches ordonnancées par *Deadline Monotonic* de traverser, de manière asynchrone et en conservant leur périodicité, les différents modes d'un système. Le protocole établit les règles suivantes pour les classes des tâches mentionnées ci-dessous :

- *Les tâches d'ancien mode* peuvent toutes s'achever indifféremment au changement de mode en cours.
- La possibilité *d'abandonner* une tâche d'ancien mode au moment du changement de mode n'est pas prévue.
- Vu que le protocole préserve la périodicité, *les tâches inchangées* continuent à s'exécuter sans tenir compte de la transition de modes.
- Par contre, *les tâches changées* modifient leur schéma d'exécution de telle sorte que la nouvelle version est toujours déclenchée après la fin de la période de l'ancienne.
- *Les tâches entièrement nouvelles* sont introduites avec un délai $Y_i \geq 0$ après l'instant de changement de mode.

Pour les besoins de l'analyse d'ordonnançabilité, le protocole ne distingue que deux groupes de tâches : les tâches changées et les tâches entièrement nouvelles. Les tâches changées présentent une nouvelle et une ancienne version. Une tâche qui existe uniquement dans l'ancien mode peut être modélisée comme une tâche changée dont le temps d'exécution de la nouvelle version est égal à zéro. De même une tâche inchangée peut être représentée par une tâche changée dont l'ancienne et la nouvelle version sont identiques.

L'analyse d'ordonnancement pendant le changement de mode est fondée sur l'approche présentée par Audsley [14] (voir section 2.4.1). Les interférences que subissent des groupes de tâches particuliers dans un intervalle de temps donné sont examinées en définissant des

fenêtres temporelles qui reflètent la phase transitoire. Cela permet de prendre en compte les interférences provenant de deux modes différents se succédant et d'avoir une perspective assez large pour repérer le pire cas possible. Le calcul de la fenêtre repose sur une procédure itérative commençant avec une taille initiale de celle-ci mise à nulle. Il s'achève lorsque la longueur de la fenêtre n'augmente plus avec les prochaines itérations ou lorsque la charge des interférences accumulées au sein de cette fenêtre est si importante que le dépassement de l'échéance de la tâche examinée peut être constaté.

Les auteurs admettent que leur analyse contient un certain degré de pessimisme et que, par conséquent, elle doit être considérée comme une condition suffisante mais pas nécessaire. Nous présentons les analyses d'ordonnançabilité pour *une ancienne version de tâche changée*, *une nouvelle version de tâche changée* et pour *une tâche entièrement nouvelle*.

L'indice i sera employé pour désigner la tâche dont on cherche le pire temps de réponse tandis que l'indice j désigne celles qui ont des priorités plus élevées que i.

Une ancienne version de tâche changée Afin de trouver le pire temps d'exécution de la tâche τ_i une fenêtre W_i qui s'étend de la date d'activation r_i de celle-ci jusqu'à sa fin f_i, est analysée (Figure 2.12). Le changement de mode a lieu après x unités de temps (tel que : $r_i \leq r_i + x \leq f_i$). Dans le cadre de cette fenêtre temporelle, l'exécution de la tâche τ_i peut être préemptée par les tâches changées τ_j et les tâches entièrement nouvelles, de plus haute priorité que celle-ci.

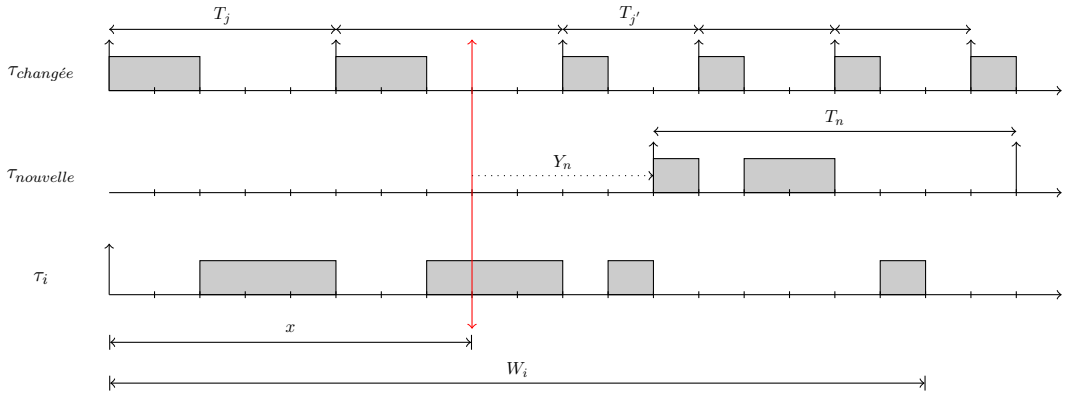


FIGURE 2.12 – Fenêtre W_i du protocole de Tindell, Burns et Wellings

En ce qui concerne l'impact des tâches changées, selon le rapport entre la tâche τ_i et les deux versions de la tâche τ_j, trois cas particuliers peuvent se produire :

(1) seule l'ancienne version de la tâche τ_j est prioritaire,

(2) seule la nouvelle l'est,

(3) tous les deux le sont.

Dans ce dernier cas, deux situations doivent encore être considérées. Le pire cas se produit soit lorsque l'ancienne version partage l'instant critique avec la tâche τ_i (Figure 2.13) soit lorsque la nouvelle est déclenchée immédiatement après le changement de mode (Figure 2.14). Seul celui de ces deux cas qui engendre la plus grande interférence est considéré dans l'analyse.

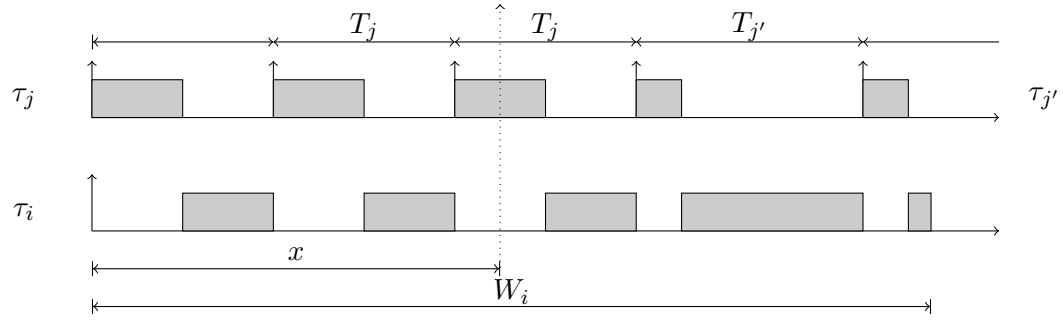


FIGURE 2.13 – L'ancienne version de tâche τ_j est déclenchée au même moment que la tâche τ_i

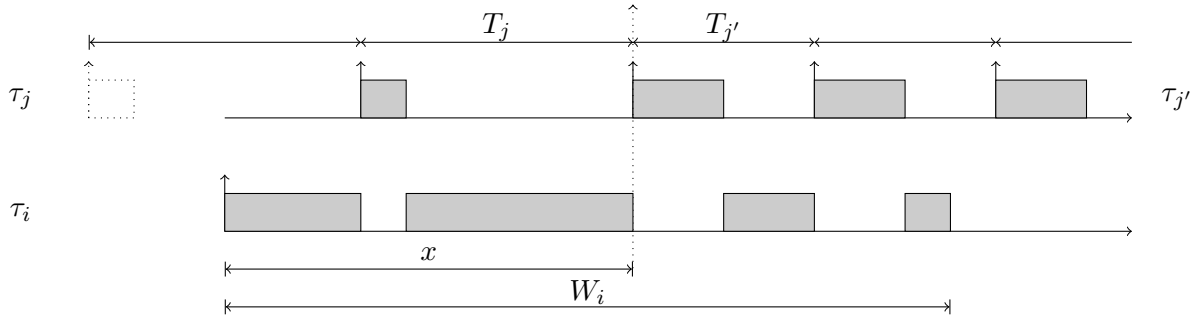


FIGURE 2.14 – Le déclenchement de la nouvelle version de tâche τ_j et le changement de mode se produisent simultanément

Les interférences provenant de tâches entièrement nouvelles sont prises en compte à partir des instants d'activation de celles-ci.

Une nouvelle version de tâche changée Pour évaluer le pire temps de réponse d'une nouvelle version de tâche changée τ_i, deux fenêtres doivent être considérées : une qui commence à l'instant du changement de mode (Figure 2.15) et l'autre qui recouvre l'ancienne et la nouvelle version de la tâche τ_i (Figure 2.16).

La première fenêtre, dénommée W^{MC}, correspond au cas dans lequel la nouvelle version de la tâche τ_i est déclenchée en même temps que le changement de mode se produit. La tâche τ_i peut partager son instant critique avec la nouvelle version de la tâche de plus haute priorité τ_j ou alors avec la dernière instance de l'ancienne version de celle-ci activée immédiatement avant le changement de mode. La figure 2.15 illustre ces deux cas. Celui des deux qui génère la charge la plus importante est ensuite considéré.

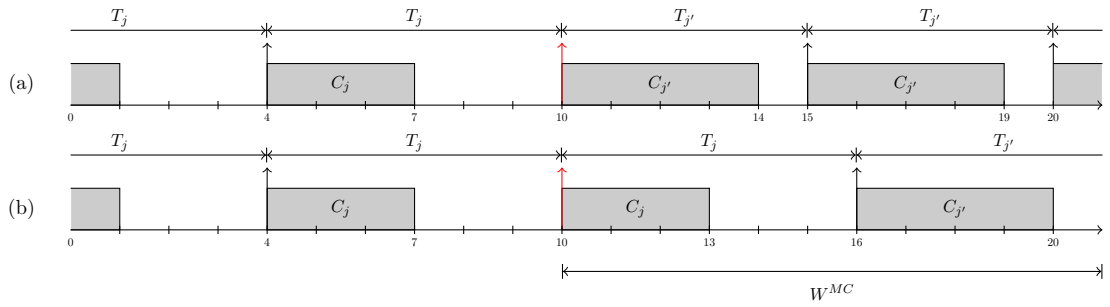


FIGURE 2.15 – (a) : La nouvelle version de la tâche τ_j est activée tout de suite après le changement de mode.
(b) : L'ancienne version de la tâche τ_j est activée juste avant le changement de mode.

La deuxième fenêtre, désignée $W_i^{OMV}(x)$, est fixée à partir de la date d'activation de l'ancienne version de la tâche changée τ_i jusqu'à la fin d'exécution de sa nouvelle version. Le changement de mode s'effectue après un certain temps x à partir du début de la fenêtre. Il est nécessaire d'analyser cette deuxième fenêtre car des tâches qui interfèrent avec l'exécution de la nouvelle version de τ_i peuvent subir l'interférence de son ancienne version (par exemple, une tâche entièrement nouvelle, activée avant que la nouvelle version de τ_i ne démarre, peut être préemptée par son ancienne version). Pour estimer la longueur de cette fenêtre, il est nécessaire de prendre en considération les temps d'exécution de l'ancienne et de la nouvelle version de τ_i ainsi que de l'interférence générée par les tâches changées τ_j et les tâches entièrement nouvelles τ_n plus prioritaires que τ_i.

Une tâche entièrement nouvelle Comme une tâche entièrement nouvelle est introduite Y_i unités de temps après un changement de mode, son ordonnançabilité peut être étudiée à l'aide de la fenêtre W_i^{MC} déjà définie pour les tâches changées dans le nouveau mode.

Protocole de Pedro et Burns Le protocole précédent permet, dans certains cas, de rendre la transition faisable en augmentant les délais après lesquels sont introduites les tâches entièrement nouvelles. Néanmoins, même en allongeant ces délais à l'infini, étant

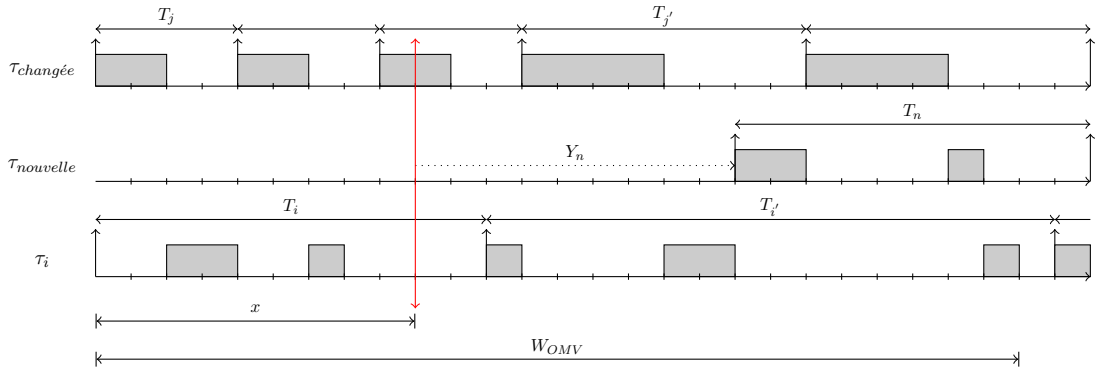


FIGURE 2.16 – La fenêtre $W_i^{OMV}(x)$

donné que les tâches changées sont contraintes par leurs anciennes et nouvelles périodes, il n'est pas toujours possible de garantir l'ordonnançabilité pour un ensemble de tâches lors d'un changement de mode. Pedro et Burns [130, 131] ont proposé un protocole dans lequel tous les types de tâches, voire les tâches changées et non-changées, peuvent être déclenchées dans un nouveau mode après un délai donné. Il est toujours possible de trouver une solution au problème de l'ordonnançabilité en choisissant des délais suffisamment larges pour ces types de tâches. Mais cela se réalise au détriment de leur périodicité. D'ailleurs, contrairement au protocole de Tindell, les auteurs admettent la possibilité d'abandonner certaines tâches d'ancien mode à l'instant d'arrivée d'une requête de changement de mode. Le protocole de Pedro et Burns peut se résumer par les points suivants :

- Les exécutions des *tâches d'ancien mode*, à l'exception des tâches abandonnées (voir le point suivant), sont menées jusqu'à leurs termes.
- *Les tâches abandonnées* s'arrêtent toutes au moment du changement de mode et sont ôtées de l'ensemble de tâches à exécuter.
- Les anciennes versions des *tâches inchangées* τ_i finissent leurs exécutions comme des tâches d'ancien mode (voir le premier point). Les nouvelles instances sont lancées Y_i unités de temps après l'instant de changement de mode. La valeur de Y_i doit être supérieure ou égale à la période de la tâche ($Y_i \geq T_i$). Ainsi, bien que tous les paramètres des tâches restent invariants, leur première instance dans le nouveau mode sera retardée ce qui entraîne une perte de périodicité.
- Les anciennes versions de *tâches changées* τ_i terminent leurs instances, pendant lesquelles intervient la demande de changement de mode, conformément aux paramètres de l'ancien mode. Ceux-ci seront modifiés dès le lancement de la première nouvelle instance Y_i unités de temps après le changement de mode. La valeur de Y_i est bornée inférieurement par la période d'ancienne version ($Y_i \geq T_{i(ancienne)}$).
- *Les tâches entièrement nouvelles* sont activées Y_i unités de temps ($Y_i \geq 0$) après le

changement de mode.

Les tâches décrites ci-dessus sont regroupées, pour les besoins de l'analyse d'ordonnançabilité, en deux classes : la classe des tâches anciennes et la classe des tâches nouvelles. La première rassemble toutes les tâches dont les instances sont activées avant le changement de mode. Elle comprend les tâches d'ancien mode ainsi que les anciennes versions des tâches changées et inchangées. Les instances déclenchées après le changement de mode, tâches entièrement nouvelles mais aussi nouvelles versions des tâches changées et inchangées, sont considérées comme des tâches nouvelles. Quoique le calcul des pires temps de réponse soit plus simple que celui du cas précédent, l'analyse n'évite pas une certaine surestimation et propose ainsi une condition suffisante mais pas nécessaire. Voici les points les plus importants concernant l'analyse *des tâches anciennes* et *des tâches nouvelles*.

La notion introduite auparavant, qui désigne par un indice i la tâche dont le pire temps de réponse est évalué et par des indices j celles qui sont plus prioritaires, sera conservée tout au long de cette section.

Tâche ancienne Pour estimer le pire temps de réponse d'une tâche ancienne τ_i, on considérera une fenêtre W_i. Elle prend son origine au moment de l'activation de τ_i et se termine à sa fin d'exécution. L'instant du changement de mode survient x unités de temps après l'origine de W_i. Pendant l'intervalle de temps défini par W_i, la tâche τ_i d'ancien mode peut subir des interférences de tâches plus prioritaires de trois types : des tâches anciennes terminées qui ont été lancées avant le changement de mode (dont les anciennes versions de tâches changées et inchangées), des tâches abandonnées à cet instant-là et toutes les tâches nouvelles (dont les nouvelles versions des tâches changées et inchangées) introduites Y_i unités de temps après x (voir Figure 2.17). Concernant les dernières instances des tâches abandonnées coïncidant avec l'instant du changement de mode, il se peut que certaines d'entre-elles soient avortées sans avoir réalisé la totalité de leur travail. Il est difficile d'évaluer précisément cette quantité sans connaître le schéma d'exécution exact de toutes les tâches. On suppose, dans le pire cas, que le temps d'exécution de cette instance ne dépasse ni son pire temps d'exécution ni l'intervalle séparant sa date d'activation et la date du changement de mode (voir Figure 2.18).

$$\sum_{j \in abandonnées} \min\left(x - \left\lfloor \frac{x}{T_j} \right\rfloor T_j, C_j\right) \tag{2.26}$$

Néanmoins, cette supposition s'avère pessimiste car elle considère que la dernière instance de la tâche abandonnée est, à chaque instant, éligible (le temps du processeur lui est toujours accordé), sans tenir compte des exécutions de tâches plus prioritaires.

Tâche nouvelle Le pire temps de réponse d'une nouvelle tâche τ_i est déterminé à l'aide d'une fenêtre W^{MC} qui s'étend, comme dans le protocole précédent, depuis l'instant de changement de mode jusqu'à la fin d'exécution de τ_i. Les dernières instances des tâches

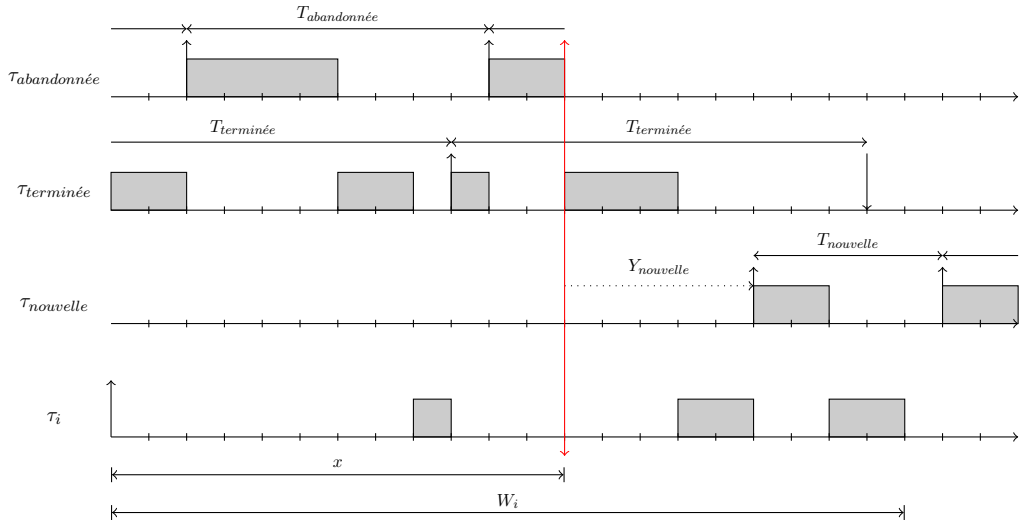


FIGURE 2.17 – Fenêtre W_i et interférences des tâches abandonnées, terminées et nouvelles sur la tâche τ_i

terminées de plus haute priorité ainsi que de nouvelles tâches qui peuvent préempter la tâche τ_i sont à l'origine d'interférences dont la présence doit être prise en compte pour estimer la longueur de la fenêtre (voir Figure 2.19).

Protocole de Real et Crespo Les deux derniers protocoles montrent qu'il existe une divergence entre exigence de périodicité et souplesse avec laquelle l'ordonnançabilité de la transition est mise au point. Dans le protocole de Pedro, toutes les tâches appartenant à la classe des tâches inchangées sont lancées pour la première fois dans le nouveau mode avec des délais, ce qui perturbe leur périodicité, mais permet d'assurer leur ordonnançabilité. Le protocole de Tindell conserve au contraire la périodicité de chacune de ces tâches mais ne laisse aucune marge de manœuvre pour assurer l'ordonnançabilité lorsque la transition n'est pas faisable. Le protocole de Real et Crespo [143], présenté ci-dessous, essaie de réconcilier les atouts des protocoles précédents en permettant que, dans un ensemble de tâches inchangées, cohabitent des tâches qui gardent leur périodicité et d'autres qui y renoncent afin de garantir l'ordonnançabilité. Ainsi, la classe des tâches inchangées comporte des tâches qui s'exécutent en respectant toujours l'exigence de périodicité et d'autres dont la première exécution, après l'instant de changement de mode, est retardée par un délai. Les détails du protocole sont les suivants :

- *Les tâches d'ancien mode*, à l'exception des tâches abandonnées (voir ci-dessous), s'achèvent normalement.
- Après l'instant de changement de mode, *les tâches abandonnées* ne sont plus exécutées.

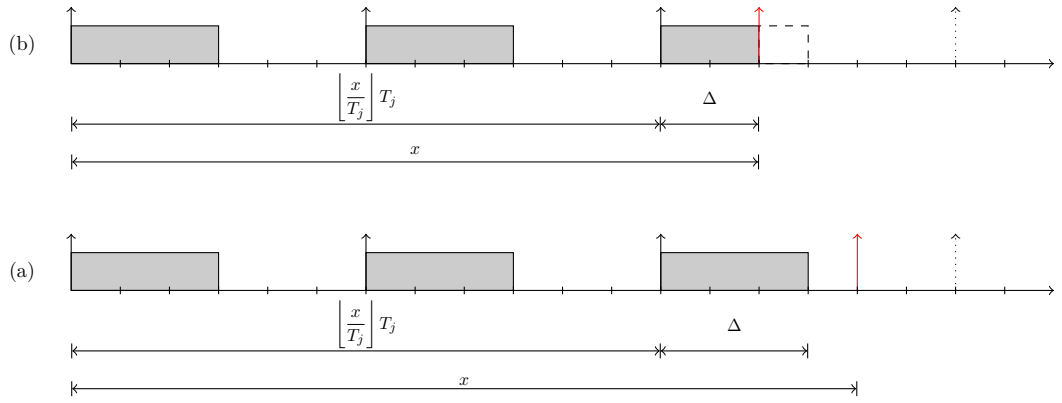


FIGURE 2.18 – (a) : La tâche abandonnée arrête son exécution au moment du changement de mode.
(b) : Le changement de mode n'impacte pas l'exécution de la tâche abandonnée.

- Une nouvelle version de *tâche inchangée* est introduite Z_i unités de temps après la fin de la période de son ancienne version. Le délai Z_i peut prendre des valeurs égales ou supérieures à zéro. Dans le cas où Z_i est égal à zéro, la périodicité de la tâche est préservée.
- Par analogie au protocole de Pedro, une nouvelle version de *tâche changée* est activée Y_i unités de temps après le changement de mode. La valeur de Y_i n'est pas inférieurement bornée par la période de son ancienne version, mais est censée être suffisamment grande pour assurer la cohérence de la transition : en particulier le cas où la nouvelle version entame son exécution avant que l'ancienne ne termine la sienne doit être évité. Il faut également noter que le changement de mode interrompt l'exécution de l'ancienne version dont la partie restant à exécuter est immédiatement abandonnée [1].
- *Les tâches entièrement nouvelles* sont activées avec un délai Y_i après l'instant de changement de mode.

Selon la valeur de Z_i, le protocole peut être considéré comme asynchrone avec périodicité ($\forall i : Z_i = 0$) ou asynchrone sans périodicité ($\exists i : Z_i > 0$). Le pire temps de réponse d'une tâche τ_i en présence de différentes tâches de plus haute priorité τ_j est recherché.

Ancienne tâche Le pire temps de réponse d'une ancienne tâche τ_i, pendant le changement de mode qui a lieu à l'instant x, est examiné au moyen d'une fenêtre W_i. Celle-ci

1. Si l'achèvement de l'ancienne version est préféré, il est toujours possible de modéliser ce comportement en fusionnant une tâche ancienne terminée avec une tâche entièrement nouvelle.

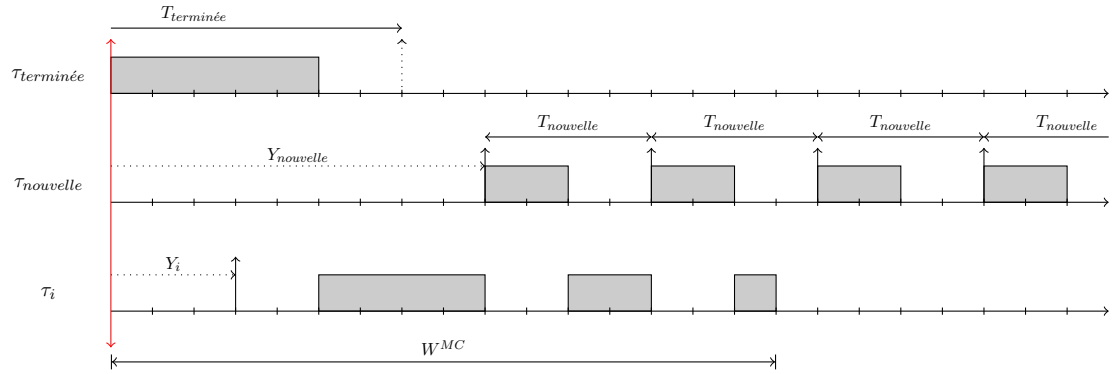


FIGURE 2.19 – Exécution d'une nouvelle tâche τ_i en présence d'une ancienne tâche terminée $\tau_{terminée}$ et d'une tâche nouvelle $\tau_{nouvelle}$.

(Figure 2.20) encadre l'exécution de la tâche τ_i depuis son activation jusqu'à sa terminaison. Les tâches de l'ancien mode (tâches terminées, tâches abandonnées ainsi que tâches inchangées et tâches changées qui poursuivent dans le nouveau mode) et celles déclenchées pour la première fois dans celui-ci (tâches entièrement nouvelles) doivent être comptées parmi les interférences que subit la tâche τ_i si elles préemptent cette tâche. Le protocole de Real et Crespo définit le comportement des tâches anciennes terminées, anciennes abandonnées, changées et entièrement nouvelles de la même manière que le protocole de Pedro. Leurs interférences peuvent donc être décrites à l'aide des mêmes méthodes. Quant aux tâches inchangées, comme leurs nouvelles versions peuvent être lancées avec des délais, ces derniers doivent être incorporés dans le calcul des interférences.

Nouvelle tâche Le temps de réponse d'une nouvelle tâche τ_i, lancée Y_i unités de temps après le changement de mode, est examiné en analysant les interférences dans la fenêtre W^{MC} qui s'étend de l'instant du changement de mode à la fin de l'exécution de τ_i (Figure 2.21). Ces interférences peuvent provenir des dernières instances des tâches anciennes terminées, qui, dans le pire cas, peuvent être déclenchées juste avant l'instant du changement de mode, des nouvelles versions des tâches inchangées, des nouvelles versions des tâches changées ainsi que des tâches entièrement nouvelles.

Les interférences dues aux nouvelles versions de tâches changées et des tâches entièrement nouvelles, car leur modèle de transition n'est pas modifié, sont traitées comme dans l'analyse d'une tâche ancienne du protocole de Pedro.

L'interférence produite par des tâches inchangées, dont le schéma de transition se distingue des autres protocoles, demande sa propre analyse. Tout d'abord, comme illustré par la Figure 2.22, il faut examiner deux cas :

(1) la dernière instance de la tâche τ_i est lancée exactement au moment du change-

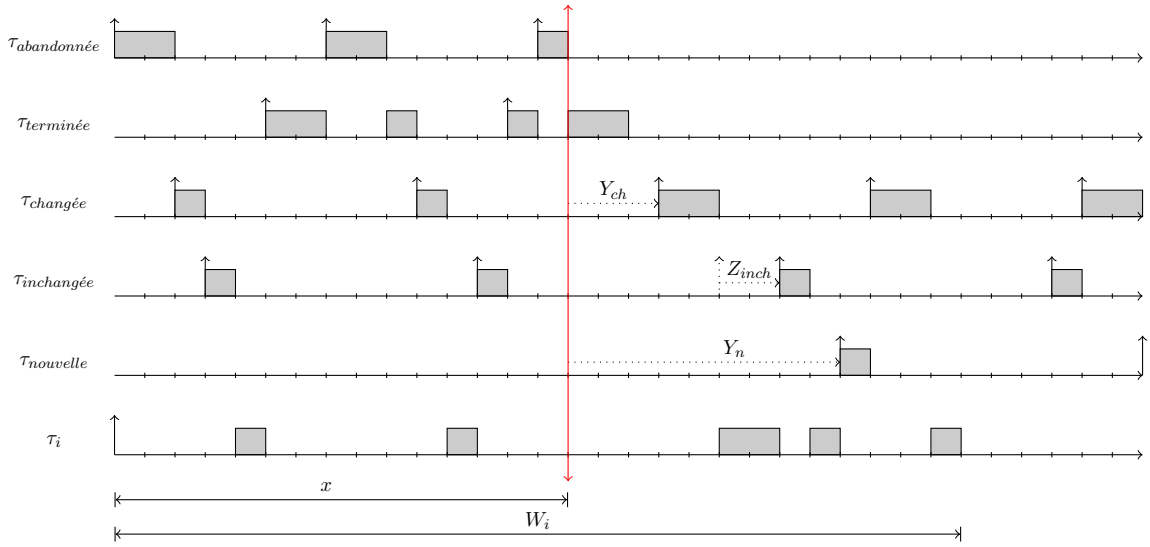


FIGURE 2.20 – La fenêtre W_i encadrant l'exécution d'une ancienne tâche τ_i en présence d'une ancienne tâche terminée $\tau_{terminée}$, d'une ancienne tâche abandonnée $\tau_{abandonnée}$, d'une tâche inchangée $\tau_{inchangée}$, d'une tâche changée $\tau_{changée}$ et d'une tâche entièrement nouvelle $\tau_{nouvelle}$.

ment de mode ; cela implique que la première instance dans le nouveau mode sera activée Z_i unités de temps après la fin de la période de l'antérieure,

(2) le déclenchement de la première instance de la tâche τ_i dans le nouveau mode Z_i unités de temps après l'instant du changement de mode.

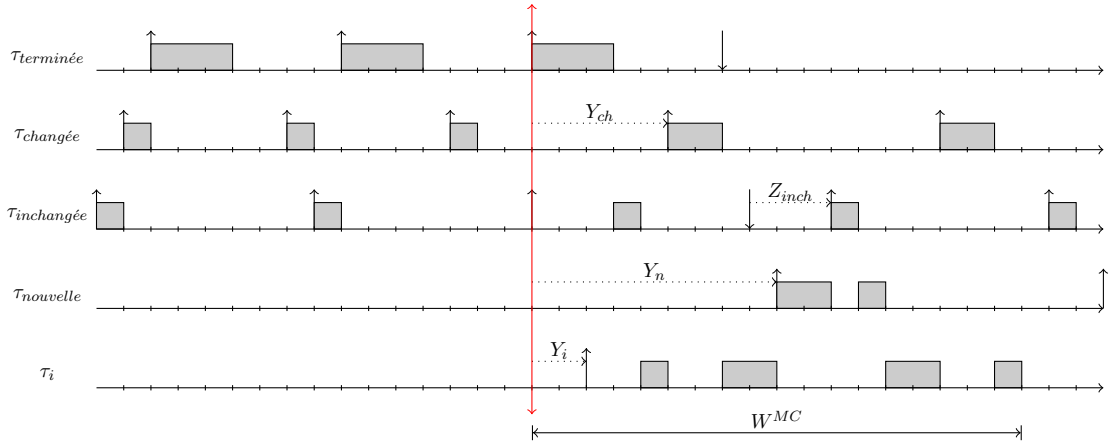


FIGURE 2.21 – Fenêtre W^{MC} encadrant l'exécution d'une nouvelle tâche τ_i activée avec un délai Y_i en présence de la dernière instance d'une ancienne tâche terminée $\tau_{terminée}$, une tâche inchangée $\tau_{inchangée}$, une tâche changée $\tau_{changée}$ et une tâche entièrement nouvelle $\tau_{nouvelle}$.

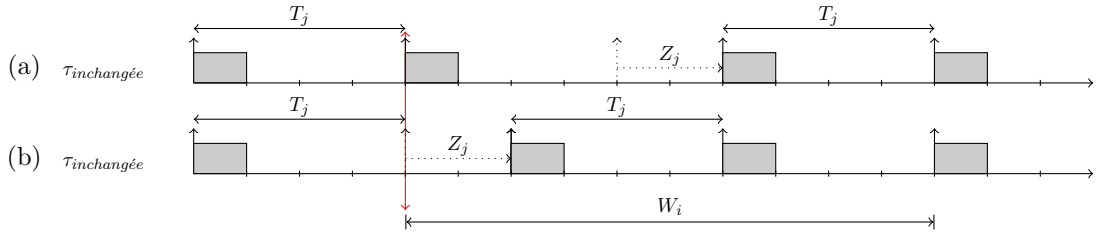


FIGURE 2.22 – (a) : La dernière instance dans l'ancien mode de la tâche $\tau_{inchangée}$ est déclenchée à l'instant du changement de mode $x = 4$.
(b) : La première instance dans le nouveau mode de la tâche $\tau_{inchangée}$ est déclenchée Z_i unités de temps après la fin de la période de la dernière instance invoquée dans l'ancien mode.

2.5.2.2 Priorité dynamique

La *demande processeur* [24] (voir section 2.4.2) paraît être un choix pertinent pour l'analyse de la phase transitoire d'un changement de mode asynchrone sous *EDF*. Plusieurs questions se posent cependant, dont celle de la détermination de la durée de l'intervalle d'étude. D'autre part le taux d'utilisation, ne permettant pas de garantir l'ordonnançabilité dans le cadre de protocoles de changement de mode avec des priorités fixes, peut permettre de constituer dans le cas *d'EDF*, sous certaines conditions, un test que sa simplicité rend applicable en ligne.

Protocole d'Andersson Reprenant l'idée du protocole de Sha [151] (début de la section 2.5.2.1), Björn Andersson [11] propose un test d'ordonnançabilité pour les systèmes de tâches utilisant le même mécanisme du changement de mode mais sous *EDF*. Si la demande du changement de mode est observée à l'instant tr, l'ancienne version d'une tâche τ_i^1 ne peut être remplacée par une nouvelle version τ_i^2 qu'à la fin de sa période T_i^1. Cet instant désigné par $transition_i$ est tel que : $tr \leq transition_i < tr + T_i^1$. Un système de n tâches est ordonnançable pendant un changement de mode se produisant à l'instant tr, si dans tous les intervalles $\Delta = [t_0, t_1]$ tels que $t_0 \leq t_r \leq t_1$, la demande de ressource processeur ne dépasse pas sa capacité :

$$\sum_{i=1}^{n} \left(\left\lfloor \frac{transition_i - t_0}{T_i^1} \right\rfloor C_i^1 + \left\lfloor \frac{t_1 - transition_i}{T_i^2} \right\rfloor C_i^2 \right) \leq \Delta \tag{2.27}$$

La longueur des intervalles à vérifier est limitée par la borne :

$$\frac{\sum_{i=1}^{n} C_i^1}{1 - \max\left(\sum_{i=1}^{n} \frac{C_i^1}{T_i^1}, \sum_{i=1}^{n} \frac{C_i^2}{T_i^2} \right)} < \Delta \tag{2.28}$$

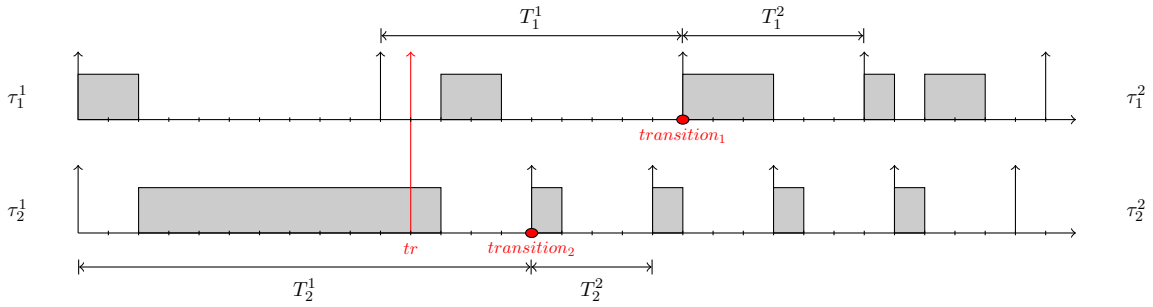


FIGURE 2.23 – Changement de mode dans le protocole d'Andersson

Protocole proposé Le protocole, pour lequel est proposé le test ci-dessous, est relativement simple. Suite à une demande de changement de mode :

- les tâches entièrement nouvelles (*nouvelles*) commencent à s'exécuter sans délai,
- les tâches d'ancien mode (*anciennes*) achèvent leurs dernières instances.

Dans le protocole d'Andersson le groupe des tâches entièrement nouvelles ou celui des anciennes complétées ne sont pas explicitement considérés. Les tâches de ces groupes, comme dans certains protocoles asynchrones avec priorité fixe, peuvent être représentées en choisissant judicieusement les paramètres de leurs anciennes (C_i^1, T_i^1) et nouvelles (C_i^2, T_i^2) versions. En considérant seulement ces deux groupes de tâches (pas de tâches changées) la complexité du test d'ordonnançabilité peut être significativement réduite. Pour la clarté de l'analyse, les tâches inchangées ne sont pas considérées mais celles-ci peuvent être introduites plus tard sans grande modification.

L'hypothèse préalablement formulée est que le démarrage synchrone de toutes les tâches anciennes et nouvelles à l'instant du changement de mode constitue le pire cas. Si l'ordonnançabilité du système à cet instant peut être prouvée, alors celui-ci est ordonnançable pour tout autre instant du changement de mode.

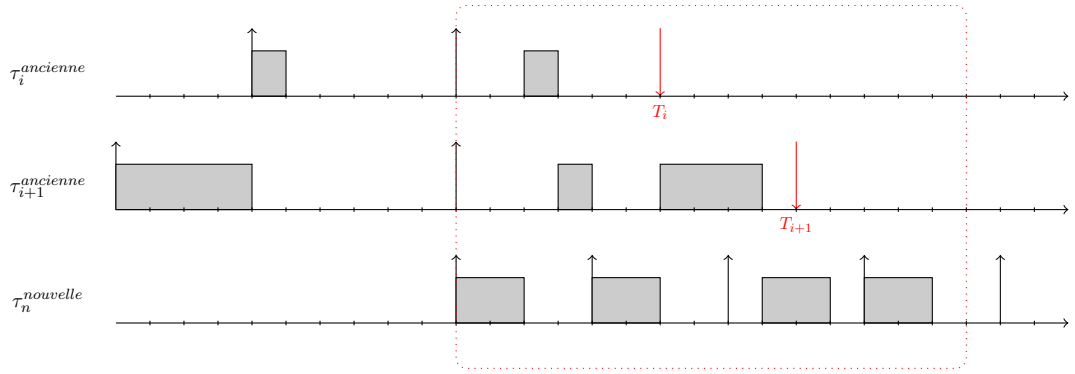


FIGURE 2.24 – Changement de mode dans le protocole proposé

L'analyse du pire cas (Figure 2.24) revient de facto à étudier l'ordonnançabilité d'un ensemble de tâches composé de tâches apériodiques (les dernières instances des tâches d'ancien mode) et de tâches périodiques (les tâches de nouveau mode). En suivant le même raisonnement que pour l'analyse de tâches synchrones (voir section 2.4.2 ou [24]), l'ensemble de tâches est ordonnançable si dans chaque intervalle $[0, t]$, où t est le temps écoulé depuis le changement de mode, les ressources du processeur demandées par les tâches d'ancien et de nouveau mode ne dépassent pas le temps de processeur disponible dans cet intervalle.

$$\sum_{i \in nouvelles} \left\lfloor \frac{t}{T_i} \right\rfloor C_i \quad + \sum_{i \in anciennes} \mathcal{H}(t - T_i) \cdot C_i \ \leq \ t \tag{2.29}$$

où $\mathcal{H}$ est une fonction de Heaviside définie par :

$$\mathcal{H}(x) = \begin{cases} 0, & \text{si } x < 0 \\ 1, & \text{si } x \geq 0. \end{cases} \tag{2.30}$$

En supposant que l'on ordonne les périodes de n tâches d'ancien mode par ordre croissant et que T_k soit la k-ième plus longue période. Pour tout t dans l'intervalle $[T_k, T_{k+1}]$, l'inégalité 2.29 se réécrit comme :

$$\sum_{i \in nouvelles} \left\lfloor \frac{t}{T_i} \right\rfloor C_i \quad + \sum_{\{i \in anciennes : T_i \leq T_k\}} C_i \ \leq \ t \tag{2.31}$$

Si la condition suivante est vérifiée :

$$\sum_{i \in nouvelles} U_i \quad + \quad \frac{\sum_{\{i \in anciennes : T_i \leq T_k\}} C_i}{T_k} \ \leq \ 1 \tag{2.32}$$

alors, les termes de la partie gauche majorant ceux de la partie gauche de 2.31, l'ordonnançabilité dans l'intervalle $[T_k, T_{k+1}]$ est vérifiée.

Cette relation doit être étendue à tous les intervalles $[0, T_1], [T_1, T_2], \ldots, [T_n, \infty]$ pour que l'ordonnançabilité du système puisse être prouvée.

$$\sum_{i \in nouvelles} U_i \quad + \quad \max_k \left\{ \frac{\sum_{\{i \in anciennes : T_i \leq T_k\}} C_i}{T_k} \right\} \ \leq \ 1 \tag{2.33}$$

Exemple 2.5. *Soit un système opérant dans un des deux modes, ancien ou nouveau, dont les tâches sont résumées dans le tableau ci-dessous.*

tâche	C_i	T_i	*mode*
τ_1	3	8	*ancien*
τ_2	2	6	*ancien*
τ_3	1	3	*nouveau*

TABLE 2.3 – Tâches d'ancien et de nouveau mode

Les taux d'utilisation du processeur de chacun de ces deux modes s'élèvent respectivement à $U_{ancien} = \frac{17}{24} \approx 71\%$ *pour l'ancien mode, et à* $U_{nouveau} = \frac{1}{3} \approx 33\%$ *pour le nouveau mode, leur somme étant supérieure à* 1. *Pour vérifier l'ordonnançabilité de ce système pendant le changement de mode, la condition 2.33 doit être appliquée :*

$$\frac{1}{3} + \max\left\{\frac{2}{6}, \frac{5}{8}\right\} = \frac{23}{24} \leq 1$$

La Figure 2.25 confirme la thèse de l'ordonnançabilité du système étudié dans cet exemple.

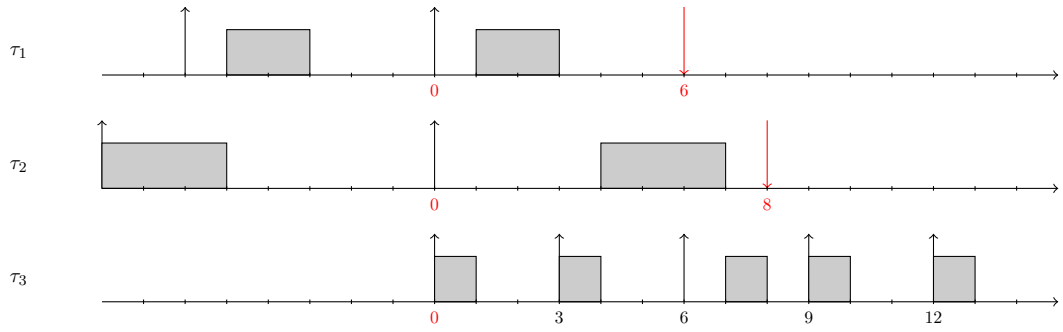


FIGURE 2.25 – Exemple d'un système à deux modes avec une transition opérée selon le protocole proposé

La formule 2.33 fournit une condition seulement suffisante. Si elle ne peut être vérifiée sur l'un des intervalles définis par les périodes des tâches d'ancien mode, il est alors nécessaire de tester la condition 2.29 donnée au départ.

Afin de limiter la durée maximale des intervalles qui doivent être analysés, une borne de faisabilité doit être déterminée. La borne de faisabilité est obtenue en appliquant le même raisonnement que celui proposé par Baruah dans le travail concernant *Recurring Branching* [25] (formule 2.21 dans la section 2.4.2). S'il existe un intervalle tel que le temps du processeur demandé par des tâches qui doivent s'exécuter dans ses limites dépasse sa longueur, le système est infaisable :

$$\sum_{i \in nouvelles} \left\lfloor \frac{t}{T_i} \right\rfloor C_i \quad + \sum_{i \in anciennes} \mathcal{H}(t - T_i) \cdot C_i \quad > \quad t \tag{2.34}$$

Dans ce cas la condition suivante est vérifiée :

$$\sum_{i \in nouvelles} \frac{t}{T_i} C_i \quad + \sum_{i \in anciennes} C_i \quad > \quad t \tag{2.35}$$

Cela donne une borne de faisabilité :

$$\frac{\sum_{i \in anciennes} C_i}{1 - \sum_{i \in nouvelles} U_i} > t \tag{2.36}$$

Tâche élastique Le fonctionnement d'un système affecté momentanément d'une charge excessive, impossible à réaliser sans pénaliser les activités et les calculs en cours, peut se dégrader au point de ne plus respecter ses échéances temporelles. Buttazzo et al. [37, 42, 40] proposent un modèle de tâche temps réel dont le taux d'utilisation s'adapte en fonction de l'accessibilité aux ressources processeur, réduisant sa consommation en ressources lorsqu'elles ne sont pas suffisantes et l'augmentant lorsqu'elles le sont. Par analogie avec la loi élastique

linéaire énoncée par Robert Hooke, une tâche $\tau_i(C_i, T_{i_0}, T_{i_{min}}, T_{i_{max}}, e_i)$ est caractérisée par trois périodes (nominale T_{i_0}, minimale $T_{i_{min}}$, maximale $T_{i_{max}}$) et un coefficient d'élasticité e_i. La période courante ainsi que le taux d'utilisation U_i de la tâche dépendent de e_i. La période de la tâche peut ainsi varier entre $T_{i_{min}}$ et $T_{i_{max}}$, ce qui permet d'ajuster dynamiquement son taux d'utilisation et d'en attribuer une partie à des tâches nouvelles. Le coefficient e_i détermine son degré d'adaptabilité. Les auteurs présentent un algorithme qui, étant donné l'abaissement demandé du taux d'utilisation de l'ensemble des tâches, calcule de combien la période de chaque tâche peut être rallongée. Les tâches, dès que le système sort d'une phase du surcharge, reviennent à leurs périodes et à leurs taux d'utilisation nominaux.

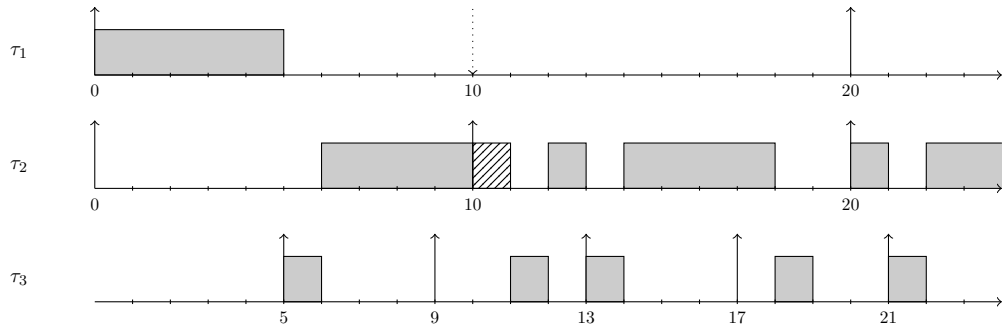
Lorsqu'arrive une demande de réduction du taux d'utilisation des tâches du système, leurs périodes sont, selon cet algorithme, immédiatement prolongées et le taux d'utilisation s'abaisse en proportion de cet allongement. Les tâches nouvelles disposent donc du taux d'utilisation libéré par ces tâches. Néanmoins, cette ressource n'est pas disponible immédiatement comme le montre l'exemple suivant [42].

Exemple 2.6. *Soit un ensemble de tâches dont les caractéristiques sont donnés dans le Tableau 2.4.*

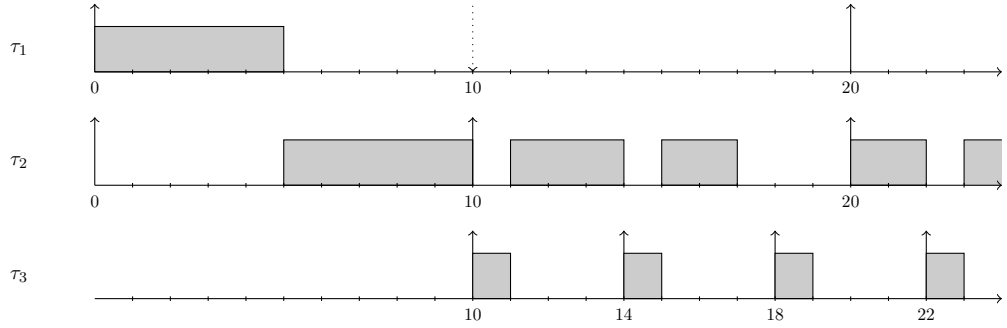
tâche	C_i	T_{i_0}	$T_{i_{min}}$	$T_{i_{max}}$	e_i
τ_1	5	10	10	20	1
τ_2	5	10	10	10	0
τ_3	1	4	4	4	0

TABLE 2.4 – Ensemble de tâches avec une tâche élastique

Initialement, les tâches τ_1 et τ_2 s'exécutent. En supposant qu'à l'instant 5, la tâche τ_3 doive intégrer l'ensemble des tâches. Le taux d'utilisation est égal à 1 ce qui ne permet pas que la tâche τ_3, ayant comme taux d'utilisation du processeur 0.25, soit insérée dans cet ensemble sans modifier les paramètres des tâches τ_1 et τ_2. La tâche τ_1 est la seule dont la période peut être rallongée. Celle-ci est donc fixée à 20 et le taux d'utilisation de τ_1 diminue de 0.25. Cette diminution peut ensuite être allouée à la tâche τ_3. La question est de savoir à partir de quand les ressources libérées sont accessibles. Comme illustré par la Figure 2.26(a), si la tâche τ_3 démarre à l'instant 5, la tâche τ_2 dépasse son échéance. En effet, 5 unités de temps du processeur ont été accordées à la tâche τ_1 sur l'intervalle de 10 unités et à l'instant de l'insertion de la nouvelle tâche, elles sont déjà consommées. Pour cette raison, c'est à partir de l'instant 10 que les ressources sont libérées et que la tâche τ_3 peut en profiter pleinement comme montré à la Figure 2.26(b).



(a) Activation d'une tâche nouvelle à l'instant 5



(b) Activation d'une tâche nouvelle à l'instant 10

FIGURE 2.26 – Insertion d'une tâche nouvelle dans un ensemble contenant une tâche élastique

Pour une tâche τ_i qui a été activée à l'instant r_i et qui change son taux d'utilisation de U_i à U_i' à l'instant t, après s'être exécutée $e_i(t)$ unités de temps, Buttazzo et al. [42] constatent qu'une allocation des ressources $e_i(t)$ a été faite sur l'intervalle $[r_i, \delta_i]$ tel que :

$$U_i = \frac{e(t)}{\delta_i - r_i} \tag{2.37}$$

Le taux d'utilisation $U_i - U_i'$ est disponible pour d'autres tâches à partir de l'instant δ_i :

$$\delta_i = d_i - \frac{c_i(t)}{U_i} \tag{2.38}$$

où $c_i(t) = C_i - e_i(t)$ et $d_i = r_i + T_i$ est la date d'échéance de la tâche τ_i.
Cet instant a été défini encore plus précisément en [79], permettant d'accéder aux ressources libérées avec un délai raccourci.

2.5.3 Protocoles pour multicœurs

L'arrivée des architectures multicœurs a posé de nouveaux défis à l'étude de l'ordonnançabilité. Si l'utilisation de plusieurs processeurs offre une meilleure efficacité et de meilleures performances, elle ajoute aussi une nouvelle dimension au problème de l'ordonnançabilité étudié auparavant. Les protocoles et les analyses pour le changement de mode ne considéraient jusqu'à présent qu'un seul processeur. Il n'est pas simple, compte tenu de la migration de tâches entre processeurs, de déterminer quels processeurs sont alloués aux tâches au moment du changement de mode. Les protocoles de changement de mode existants à l'heure actuelle dans le contexte multicœurs sont des extensions du protocole du délai minimal (Minimum Single Offset) présenté en section 2.5.1.2.

Nelis, Goossens et Andersson en ont proposé deux versions pour le multicœur :

- multiprocesseur synchrone délai minimal,
  Synchronous Multiprocessor Minimum Single Offset, SM-MSO [120, 122, 123],
- multiprocesseur asynchrone délai minimal,
  Asynchronous Multiprocessor Minimum Single Offset, AM-MSO [120, 123].

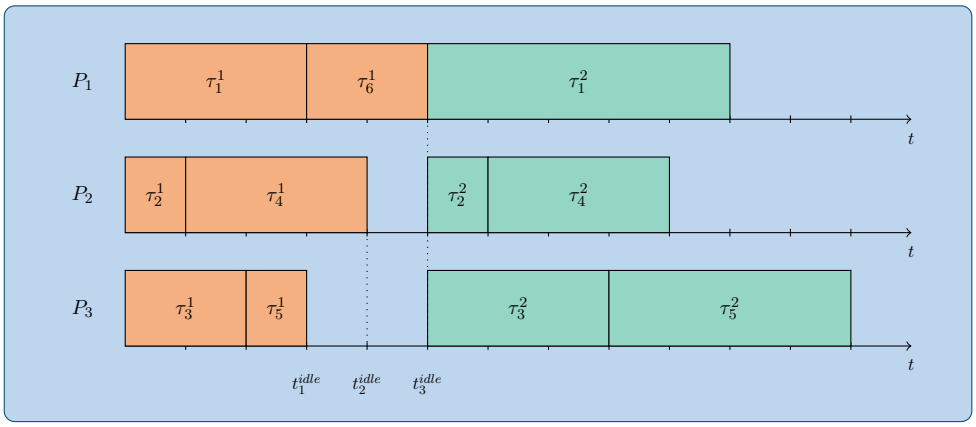


FIGURE 2.27 – Synchronous Multiprocessor Minimum Single Offset

Le principe de ces deux protocoles est le même que celui du protocole original : à partir de l'instant du changement de mode, les nouvelles instances des tâches ne sont plus activées et celles déjà en cours se terminent normalement. Les tâches de nouveau mode sont déclenchées différemment dans chacun des protocoles. Dans *SM-MSO*, elles ne sont introduites qu'après la fin de toutes les tâches de mode ancien (Figure 2.27). Dans *AM-MSO*, une première tâche de nouveau mode peut commencer son exécution sur un des processeurs libre lorsqu'il n'y a plus de tâche d'ancien mode à ordonnancer (Figure 2.28). Cependant, les auteurs reconnaissent que cette approche n'est pas toujours optimale en termes des délais observés

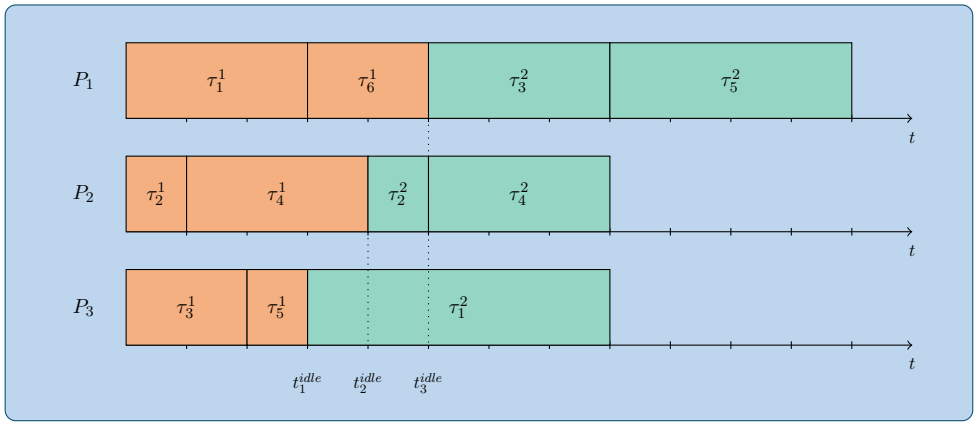


FIGURE 2.28 – Asynchronous Multiprocessor Minimum Single Offset

et un algorithme pour ordonnancer les tâches de nouveau et d'ancien mode pendant la phase transitoire, sans priorité pour ces dernières, serait à proposer.

Exemple 2.7. *Soient trois processeurs P_1, P_2, P_3 sur lesquels s'exécutent 6 tâches apériodiques : $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5$ et τ_6. Chacune de ces tâches τ_i a une version dans un des deux mode d'exécution : τ_i^1 dans le premier mode et τ_i^2 dans le second mode. Les pires temps d'exécution des tâches, selon leur mode d'exécution, sont comme suit :*

	τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
c_i^1	*3*	*1*	*2*	*3*	*1*	*2*
c_i^2	*5*	*1*	*3*	*3*	*4*	*0*

La Figure 2.27 illustre un changement de mode opéré pour cet ensemble de tâches selon le protocole SM-MSO *et, sur la Figure 2.28, selon le protocole* AM-MSO*. L'instant t_i^{idle} désigne un instant auquel i processeurs ont fini l'exécution des tâches d'ancien mode.*

L'analyse de ces deux protocoles ne porte sur aucune politique d'ordonnancement spécifique, en faisant toujours l'hypothèse de l'allocation de tâches au moment du changement de mode la pire possible. Elle cherche simplement à obtenir des bornes sur les délais après lesquels les tâches de nouveau mode sont activées. Dans ce qui suit, m représentera le nombre de processeurs et $c_1, c_2 \ldots, c_n$ les pires temps d'exécution des tâches d'ancien mode en ordre croissant ($c_1 \leq c_2 \leq \ldots \leq c_n$).

Le temps à partir duquel les nouvelles tâches sont ordonnancées dans *SM-MSO* coïncide avec la fin d'exécution de la dernière tâche de l'ancien mode. Une tâche d'ancien mode τ_i, avant le passage au nouveau mode, est activée pour la dernière fois à l'instant t_i^{alloc}. Cet instant, si $hp(\tau_i)$ désigne l'ensemble des tâches plus prioritaires que la tâche τ_i, est borné

supérieurement par la valeur de $\hat{t}_i^{alloc}$ telle que :

$$\hat{t}_i^{alloc} = \begin{cases} 0 & \text{si } m \geq n \\ \frac{1}{m} \sum\limits_{\tau_j \in hp(\tau_i)} c_j & \text{sinon} \end{cases} \tag{2.39}$$

La tâche τ_i sera finie à l'instant t_i^{comp}, c_i temps après le début de son exécution : $t_i^{comp} = t_i^{alloc} + c_i$. A l'aide des deux relations précédentes, les auteurs prouvent qu'aucune tâche τ_i ne termine son exécution après le temps $\hat{t}_i^{comp}$ (borne supérieure).

$$\hat{t}_i^{comp} = \begin{cases} c_n & \text{si } m \geq n \\ \frac{1}{m} \sum\limits_{j=1}^{n} c_j + \left(1 - \frac{1}{m}\right) c_n & \text{sinon} \end{cases} \tag{2.40}$$

C'est au plus tard à cet instant-là que les tâches de nouveau mode seront activées.

Les ressources des processeurs, dans la version asynchrone du protocole, *AM-MSO*, sont libérées graduellement. Au fil des instants t_i^{idle}, plus petit instant auquel i processeurs ont fini l'exécution des tâches d'ancien mode, les processeurs sont attribués, l'un après l'autre, aux tâches de nouveau mode. L'analyse cherche à trouver combien de temps après la requête de changement de mode au moins k processeurs sont disponibles. Elle part du constat qu'à l'instant t_1^{idle}, où un premier processeur devient libre, il n'y a plus de tâche d'ancien mode à ordonnancer sur un des $m-1$ processeurs restants. Chacun de ces processeurs exécute encore une tâche d'ancien mode. Dans le pire cas, k processeurs seront libres après le temps c_{n-m+k}. Cet instant est borné par l'expression suivante :

$$\hat{t}_k^{idle} = \begin{cases} 0 & \text{si } (n \leq m) \wedge (m - n \geq k) \\ c_{k-m+n} & \text{si } (n \leq m) \wedge (m - n < k) \\ \max\limits_{i=0}^{n-m+k-1} \left\{ \frac{\sum_{j=1}^{n} c_j - \sum_{j=i+1}^{i+m-k+1} c_j}{m} + \frac{\sum_{j=i+1}^{i+m-k+1} c_j}{m-k+1} \right\} & \text{sinon} \end{cases} \tag{2.41}$$

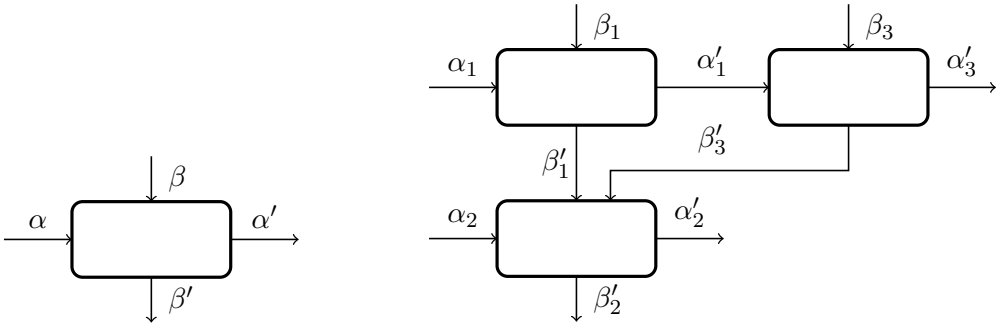
Les améliorations ultérieures apportées à ces deux protocoles s'articulent autour de trois axes. Tout d'abord, Niels et al. [121] ont intégré les tâches inchangées dans l'analyse. Ensuite ils ont réévalué les bornes avec *global EDF* comme politique d'ordonnancement. Finalement, Yomsi et al. [175] ont intégré à l'analyse des facteurs déterminant les vitesses auxquelles chacun des processeurs exécute des tâches.

2.5.4 Changement de mode dans l'analyse compositionnelle d'un système

Le besoin croissant de pouvoir assembler des systèmes à partir de différents composants, chacun réalisant une certaine fonctionnalité, afin d'obtenir, par coopération et synthèse, le comportement d'ensemble souhaité, a suscité l'émergence de plusieurs méthodes et

techniques de composition. En contexte temps-réel, le point crucial est d'assurer que les ressources allouées à chaque composant sont suffisantes pour que celui-ci remplisse sa fonction dans les délais de temps imposés. L'analyse d'ordonnançabilité d'un système conçu de manière compositionnelle évalue son fonctionnement global sur la base des informations concernant les exigences temporelles de ses composants et du schéma selon lequel les ressources leur sont allouées. Les exigences temporelles des composants ainsi que l'allocation des ressources peuvent changer de mode de fonctionnement pour mieux suivre l'évolution du processus contrôlé au sein d'un environnement dont les paramètres peuvent être sujets à de possibles variations. Les points qui suivent présentent les méthodes permettant de modéliser, dans les conditions dynamiques du changement de mode, le comportement temporel et de vérifier la faisabilité de tels systèmes, moyennant une description de leurs composants.

Real-Time Calculus Le Real-Time Calculus [169, 134, 167], qui trouve son origine dans le *Network Calculus* [104], une méthode pour analyser les flux de données dans les réseaux, propose une vision du système représenté par un ensemble de blocs interconnectés. Chaque bloc transforme des flux arrivant à ses entrées et les transfère vers d'autre blocs reliés à ses sorties. Deux types des flux sont considérés : *les flux d'événements* et *les flux de service*. Les flux d'événements α sont traités dans les blocs moyennant des ressources délivrées par des flux de service β. A la suite de ce traitement, les flux β' des ressources qui n'ont pas été utilisées ainsi que les flux α' des événements exécutés sortent de chaque bloc et peuvent rejoindre les entrées d'autres blocs.



(a) Bloc de base (b) Exemple d'un système

FIGURE 2.29 – Real-Time Calculus : du bloc de base à la composition d'un système

Si $R[s,t]$ désigne le nombre d'événements qui arrivent dans l'intervalle de temps $[s,t]$, le couple $\alpha = [\alpha^u, \alpha^l]$ décrit ce flux par la relation suivante :

$$\alpha^l(t-s) \leq R[s,t] \leq \alpha^u(t-s) \quad \forall s < t \tag{2.42}$$

$\alpha^l(\Delta)$ et $\alpha^u(\Delta)$ représentent respectivement les bornes inférieure et supérieure sur le nombre d'événements qui peuvent arriver dans n'importe quel intervalle de temps de longueur Δ.
De même, si $C[s,t]$ désigne la quantité de ressources, comme par exemple les cycles de processeur, délivrée dans l'intervalle de temps $[s,t]$, le couple $\beta = [\beta^u, \beta^l]$, qui est associé à ce flux, satisfait la relation ci-dessous :

$$\beta^l(t-s) \leq C[s,t] \leq \beta^u(t-s) \quad \forall s < t \tag{2.43}$$

$\beta^l(\Delta)$ et $\beta^u(\Delta)$ représentent respectivement les borne inférieure et supérieure sur le nombre de ressources disponibles dans n'importe quel intervalle de temps de longueur Δ.

Cette abstraction permet d'exprimer des processus d'arrivée beaucoup plus complexes que les processus périodiques. Ces derniers peuvent toujours être obtenus en appliquant les mêmes méthodes que pour le calcul de demande totale du processeur (voir formule 2.18 dans la section 2.4.2). Pour les processus d'arrivée, qui peuvent être représentés par une machine d'états finis, Wandeler, Maxiaguine et Thiele [177] proposent un algorithme qui détermine les bornes les caractérisant. Wandeler [176], sur l'exemple de deux machines d'états finis, dont l'une représente l'ordre dans lequel se font les demandes d'accès aux données et l'autre une mémoire cache qui enregistre des copies des données les plus récemment utilisées afin d'en réduire temps d'accès, montre comment coupler plusieurs processus corrélés afin d'obtenir une représentation qui tienne compte de leurs dépendances.

Cette caractérisation des flux d'événements et de services rend possible une évaluation de la performance du système. La taille des files d'attente Buf doit être plus grande que le nombre maximal d'événements qui peuvent arriver dans un intervalle de temps où les ressources nécessaires pour leur traitement sont délivrées à bas régime.

$$Buf(\alpha^u, \beta^l) = \sup_{\lambda \geq 0} \left\{ \alpha^u(\lambda) - \beta^l(\lambda) \right\} \tag{2.44}$$

Le délai Del entre l'arrivée des événements et la fin de leur traitement, dont l'estimation est si importante en contexte temps réel, peut être déterminé en comparant l'intervalle d'arrivée du nombre maximal d'événements avec l'intervalle de disponibilité minimum des ressources nécessaires à leur traitement.

$$Del(\alpha^u, \beta^l) = \sup_{\lambda \geq 0} \left\{ \inf \left\{ \tau \geq 0 : \alpha^u(\lambda) \leq \beta^l(\lambda + \tau) \right\} \right\} \tag{2.45}$$

Le calcul des délais permet de vérifier l'ordonnançabilité d'un composant (bloc). Si celui-ci exécute n tâches, les activations de chaque tâche τ_i étant assimilées ici à un flux d'événements entrants α_i, selon l'ordre donné par *Earliest Deadline First*, la condition suivante garantit son ordonnançabilité :

$$\sum_i^n \alpha_i^u(\Delta - D_i) \leq \beta^l(\Delta) \qquad \forall \Delta \in \mathbb{R}^{\geq 0} \tag{2.46}$$

La condition 2.46 ne prend pas en compte les changements de mode. Perathoner, Stoimenov et Thiele présentent [134, 167] des conditions pour le cas où un flux d'événements peut, à un certain instant, prendre une forme différente. Si Γ_I et Γ_{II} désignent respectivement l'ensemble de tâches qui doivent être terminées dans le mode I et l'ensemble de tâches activées dans le mode II, la transition entre ces deux modes est faisable si la condition suivante est vérifiée pour tous les intervalles de temps Δ :

$$\sup_{0\leq\lambda\leq\Delta}\left\{\sum_{\tau\in\Gamma_I}\alpha_\tau^u(\Delta-\max\{D_\tau,\lambda\})+\sum_{\tau\in\Gamma_{II}}\alpha_\tau^u(\lambda-D_\tau)\right\}\ \leq\ \beta^l(\Delta)\qquad\forall\Delta\in\mathbb{R}^{\geq 0}\quad(2.47)$$

La Figure 2.30 illustre la composition de flux d'événements de deux modes pendant la phase transitoire considérée dans la condition ci-dessus.

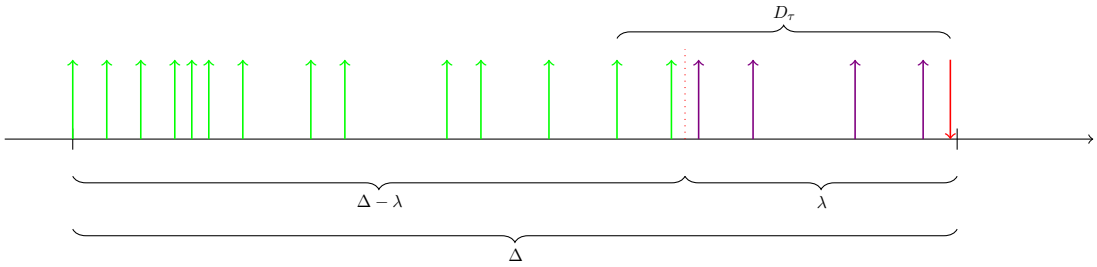


FIGURE 2.30 – Real-Time Calculus : Evénements considérés pendant le changement de mode pour EDF

Si les tâches de nouveau mode sont activées avec un délai δ, cette condition s'exprime ainsi :

$$\sup_{0\leq\lambda\leq\Delta}\left\{\sum_{\tau\in\Gamma_I}\alpha_\tau^u(\Delta-\max\{D_\tau,\lambda\})+\sum_{\tau\in\Gamma_{II}}\alpha_\tau^u(\lambda-D_\tau-\delta)\right\}\ \leq\ \beta^l(\Delta)\qquad\forall\Delta\in\mathbb{R}^{\geq 0}\quad(2.48)$$

Certains systèmes peuvent profiter temporairement d'une meilleure qualité du service ou diminuer le taux des informations traitées si, par exemple, le niveau de remplissage d'une file d'attente dépasse un certain seuil. Phan, Chakraborty et Thiagarajan [136] modélisent les flux introduits auparavant comme deux automates : un automate d'arrivée pour les événements et un automate de service pour les services. Ces deux automates passent d'un état à l'autre en changeant l'intensité d'un flux. A chaque état s sont associées les bornes d'un flux, (α_s^l, α_s^u) pour l'automate d'arrivée et (β_s^l, β_s^u) pour l'automate de service, ainsi qu'un intervalle de temps $[L_s, U_s]$ tel que L_s est le temps minimal pendant lequel l'automate doit rester dans l'état s et U_s le temps maximal pendant lequel il peut rester dans cet état.

Les transitions sont conditionnées, à la fois, par la valeur d'horloge et par le niveau de remplissage d'une file d'attente.

Cette approche propose une vision du système assez simpliste. Les événements déposés dans les files d'attente sont traités au moyen des flux de service. Cependant, un système, en fonction de son architecture et de choix de conception, gère les ressources disponibles différemment. La politique d'ordonnancement ou les paramètres des tâches, entre autres, sont des facteurs qui ont un fort impact sur la performance du système. Phan, Chakraborty et Lee prennent en considération [135] ces deux facteurs en proposant un modèle pour une application composée de plusieurs modes dont chacun est caractérisé par un ensemble de tâches, une politique d'ordonnancement (priorité fixe ou TDMA) et les bornes sur les flux α pour les événements et β pour les services. La manière dont l'application enchaîne dans le temps l'exécution successive des modes est exprimée par un automate dont les transitions dépendent de la valeur d'horloge et du niveau de remplissage de la file d'attente.

En se basant sur cette abstraction, Phan, Lee et Sokolsky [138] fondent un modèle plus polyvalent pour un système pouvant exécuter concurremment plusieurs applications, chacune dans un mode différent. L'analyse qu'ils proposent ne nécessite pas de savoir quelle est l'affectation des ressources aux composants. Les ressources nécessaires pour l'exécution de l'application sont estimées comme un résultat de l'analyse en tenant compte de toutes les demandes faites dans chacun des composants. Ce travail étend aussi le modèle précédent du point de vue du changement de mode. Dans ce contexte, les tâches complétées, nouvelles et inchangées sont prises en compte alors que ces dernières ne l'étaient pas auparavant.

Fisher et Ahmed considèrent également des tâches avortées [63, 4] en faisant une hypothèse sur la périodicité d'activation des tâches ainsi que sur l'allocation des ressources. Toutes les tâches sont sporadiques ($\tau_i = (C_i, D_i, T_i)$) et les ressources sont allouées à chaque composant du système selon *l'Explicit-Deadline Periodic Resource Model* [57, 155]. Le test proposé par Fisher et Ahmed retrace l'exécution multimodale d'un composant et vérifie son ordonnançabilité sous *EDF* en temps pseudo-polynomial. Cette complexité, moins élevée que pour les travaux cités précédemment, découle du modèle périodique des tâches et des ressources, ce qui a permis d'appuyer l'analyse en grande partie sur une borne de la fonction de demande dbf (voir section 2.4.2).

Fisher, Ahmed et Hettiarachchi ont récemment évoqué [62] le manque d'analyse de ce type pour les plateformes multicœurs où l'allocation des ressources peut s'effectuer selon la *parallel-supply function* (*PSF*) [33] ou *multiprocessor periodic resource* (*MPR*) [154].

Changement de mode d'allocation des ressources L'allocation des ressources est typiquement réalisée via le mécanisme d'un serveur. En général, un serveur est doté d'un budget Q qui est, tout au long d'une période P, mis à disposition des applications et des tâches attribuées à ce serveur. De ce point de vue, un serveur peut être considéré comme une tâche avec une période P et un pire temps d'exécution Q. Les algorithmes de serveurs sont basés sur une politique d'ordonnancement fixe ou dynamique. Parmi les premiers, les plus connus sont : *Polling Server* [38], *Deferrable Server* [168, 106] et *Sporadic*

Server [159]. Des serveurs s'exécutent sous *EDF* : *Dynamic Sporadic Server* [161], *Total Bandwidth Server* [3, 162] et *Constant Bandwidth Server* [2].

Les exigences d'adaptabilité et de flexibilité du système impose que celui soit capable d'appliquer différents schémas d'allocation des ressources. Santinelli, Buttazzo et Bini abordent ce problème [149] en définissant un changement de mode pour un serveur S comme le passage d'un ancien mode $S^I = (Q^I, P^I)$ vers un nouveau $S^{II} = (Q^{II}, P^{II})$. Les fonctions $sbf^I(t)$ et $sbf^{II}(t)$ dénotent l'approvisionnement minimal des ressources dans les modes S^I et S^{II}. A la suite d'une demande de reconfiguration reçue à l'instant t_{req}, le serveur, selon le protocole choisi, soit arrête de fournir des ressources soit continue de le faire jusqu'à l'activation du nouveau mode fixée à l'instant $t_{req} + \delta$. La fonction $sbf^T(t)$, qui caractérise le pire approvisionnement des ressources pendant le changement de mode est obtenue, dans ce premier modèle de transition, comme :

$$sbf^T(t) = \inf_{0 \leq \lambda \leq t} \left\{ sbf^I(t - \lambda - \gamma + P^I - Q^I) + sbf^{II}(\lambda + P^{II} - Q^{II}) \right\} \tag{2.49}$$

où $\gamma = t_{req} - t^{last} + \delta$ et t^{last} est l'instant initial de la dernière période dans l'ancien mode. Dans ce deuxième modèle elle est donnée par l'expression suivante :

$$sbf^T(t) = \inf_{0 \leq \lambda \leq t} \left\{ sbf^I(t - \lambda - \gamma + 2P^I - Q^I) + sbf^{II}(\lambda + P^{II} - Q^{II}) \right\} \tag{2.50}$$

2.6 Changement de mode dans un système à déclenchement temporel

Dans un système à déclenchement temporel, toutes les actions, y compris le changement de mode, sont initiées à des instants prédéfinis. Cela peut être réalisé, par exemple, au moyen de tableaux, chacun contenant des séquences d'ordonnancement pour un mode ou pour une phase transitoire entre deux modes [64]. Ces tableaux sont calculés à partir de graphes exprimant les relations de précédence entre les tâches de chaque mode et pour des phases transitoires.

Les sémantiques des langages de définition temporelles définissent des protocoles de changement de mode de manière plus générale, sans avoir recours à une spécification spéciale pour la phase transitoire. Néanmoins, certaines contraintes sont imposées sur le choix des instants de changements de mode ainsi que sur la position de départ dans le nouveau mode. On identifie deux types de protocoles. Le changement de mode dit *harmonique* ne peut s'opérer qu'aux points de temps où aucune des tâches n'est logiquement active et l'exécution du nouveau mode commence à son début. Le changement de mode *non-harmonique* peut, au contraire, se produire à des instants où certaines tâches sont en train de s'exécuter logiquement, à la condition qu'elles soient aussi présentes dans le nouveau mode. La position à partir de laquelle elles commencent leur exécution dans le nouveau mode correspond à la configuration de démarrage de ces tâches à l'instant du changement de mode.

Exemple 2.8. *Soient des modes m et m'. Le mode m exécute les tâches τ_1 et τ_2 avec les périodes $T_1 = 4$ et $T_2 = 3$. Le mode m' exécute les tâches τ_1 et τ_3 avec les périodes $T_1 = 4$ et $T_3 = 6$. Les périodes des deux modes sont fixées à 12. La Figure 2.31 illustre le déroulement d'un changement du mode m au mode m' selon deux types de protocoles : harmonique et non harmonique. Les flèches rouges lient la position dans l'ancien mode où ce changement est déclenché à la position de départ dans le nouveau mode.*

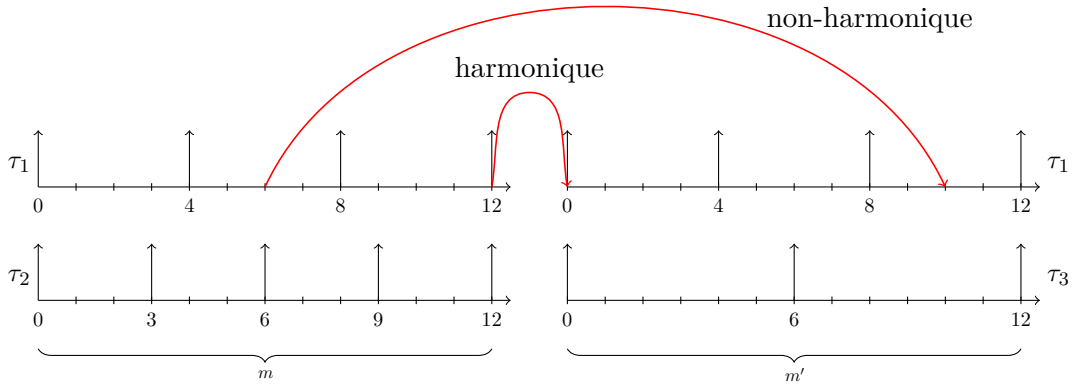


FIGURE 2.31 – Changement de mode harmonique et non-harmonique.

Le reste de cette section présente plus en détail différentes approches pour le changement de mode dans différents langages de définition temporelle comme *Giotto*, *TDL* et leurs extensions ainsi que les méthodes utilisées pour vérifier l'ordonnançabilité de ces changements de mode.

2.6.1 Giotto

Giotto implante un protocole non-harmonique du changement de mode. Pendant le changement de mode, trois types des tâches sont distingués : anciennes, inchangées et nouvelles. L'exécution logique d'aucune des tâches d'ancien mode qui ne sont pas présentes dans le nouveau mode ne doit être interrompue. Par conséquent, l'instant δ_{sw} de ce changement coïncide toujours avec un point dans le temps où toutes les tâches anciennes sont terminées, i.e un multiple commun de leurs périodes.

$$\delta_{sw} \bmod ppcm\left\{T_i | \tau_i \in \tau[m] - \tau[m']\right\} = 0 \tag{2.51}$$

Le nouveau mode m', dont la période est égale à $\pi[m']$, est immédiatement activé avec un temps de mode δ', égal à 0 dans le cas d'absence de tâches inchangées ou calculé comme

suit s'il en existe :

$$\delta' = \pi[m'] - (\epsilon - \delta_{sw}) \tag{2.52}$$

où ϵ et γ sont tels que :

$$\epsilon = \left\lceil \frac{\delta_{sw}}{\gamma} \right\rceil \gamma \quad et \quad \gamma = ppcm\left\{T_i | \tau_i \in \tau[m] \cap \tau[m']\right\}$$

Autrement dit, $(\epsilon - \delta_{sw})$ représente le temps nécessaire pour que toutes les tâches inchangées puissent terminer leur exécution logique. Le nouveau mode reprend alors juste avant la fin de la période d'ancien mode là où ces tâches sont au même stade de leurs exécutions auquel elles étaient dans l'ancien mode. Les tâches nouvelles ne seront déclenchées qu'aux dates d'activation usuelles.

Exemple 2.9. *Soit un programme Giotto composé des deux modes m_1 et m_2. Les deux modes ont des périodes égales à 120 et chacun comprend trois tâches, mode m_1 : τ_1, τ_2, τ_3 et mode m_2 : τ_1, τ_2, τ_3'. Ces tâches sont caractérisées par les temps d'exécution logique et les périodes suivantes : $\tau_1 = (15, 120), \tau_2 = (30, 60), \tau_3 = (15, 40), \tau_3' = (10, 30)$. La figure ci-dessous illustre le changement de mode qui se produit à l'instant $\delta_{sw} = 40$ dans le mode m_1. Les tâches inchangées τ_1 et τ_2 continuent leurs exécutions tandis que la tâche ancienne τ_3 est remplacée dans le nouveau mode m_2 par sa nouvelle version τ_3'.*

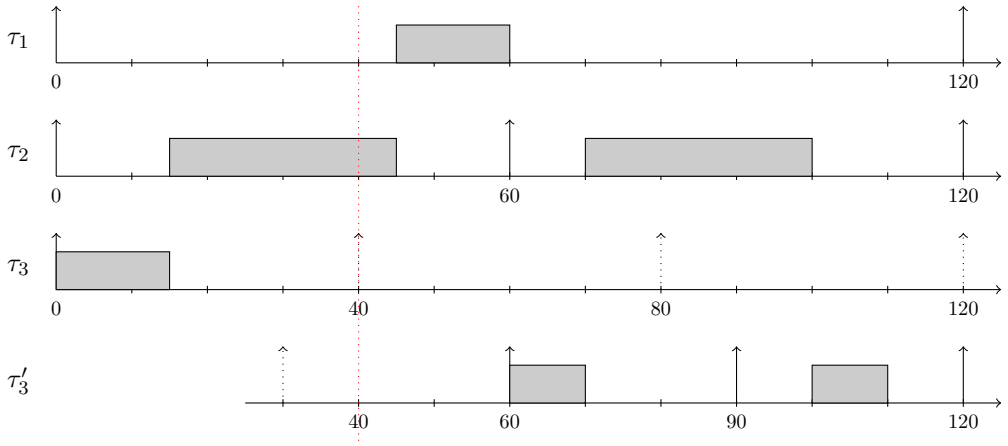


FIGURE 2.32 – Changement de mode dans Giotto

Le problème de l'ordonnançabilité dans Giotto peut se formuler comme un *jeu de sécurité* (ang. *safety game*) entre deux joueurs [88, 71]. Le premier joueur est un ordonnanceur qui choisit une tâche à exécuter au prochain tick d'horloge. Ensuite, le second joueur, l'environnement, choisit des événements qui peuvent déclencher certaines actions du système, comme un changement de mode. Le jeu continue ainsi, tour à tour, et son

déroulement peut être retracé par un graphe de tous les états possibles du système. Le but de l'ordonnanceur est de rester toujours dans des états ordonnançables tandis que l'environnement essaie d'en sortir pour rejoindre des états infaisables. Cette approche, assez coûteuse en temps, permet également de vérifier d'autres propriétés du système comme, par exemple, l'absence de conditions de concurrence.

La stratégie de l'ordonnanceur peut être associée à une politique d'ordonnancement, ce qui limite le nombre des choix potentiels pour le premier joueur. De plus, s'il est possible que chaque mode du système puisse s'exécuter sans interruption du début à la fin, l'ordonnançabilité sous *EDF* peut être vérifiée avec le test sur le taux d'utilisation [88] (voir formule 2.11). Si l'environnement ou la construction du programme ne permettent pas qu'un de ses modes s'exécute pleinement au moins une fois, toutes conditions d'environnement confondues, le test peut être considéré comme une condition suffisante mais pas nécessaire. Ghosal [72] propose dans ce cas d'appliquer le test de Horn pour des tâches apériodiques [91].

En différenciant échéance d'exécution des requêtes et date d'activation d'une prochaine requête, des plages temporelles autres que multiples des périodes de tâches anciennes, peuvent être considérées. Une tâche est donc logiquement inactive entre sa date d'échéance et sa prochaine date d'activation et un changement de mode peut alors intervenir. En s'appuyant sur cette observation, Martinek et Pohlmann [114] ont proposé trois protocoles de changement de mode pour un modèle de tâche étendu tel que $\tau_i = (TEL_i, D_i, T_i)$. Le premier, *Basic Mode Switch Protocol* (*BMSP*), permet de définir un changement de mode dans les plages d'inactivité des tâches anciennes. Le deuxième, *Extended Mode Switch Protocol* (*XMSP*) introduit un groupe de tâches avortées, ce qui assouplit encore et rend plus réactif le mécanisme de changement de mode. Enfin dans *Abortive Mode Switch Protocol* (*AMSP*) toutes les tâches d'ancien mode sont avortées dès le changement de mode. Sauf pour ce dernier protocole, l'ordonnançabilité de chaque mode du système n'implique nullement l'ordonnançabilité du système entier. Les transitions entre modes doivent être considérées afin de résoudre le problème d'ordonnançabilité. Les auteurs esquissent brièvement un test basé sur le pire temps de réponse. Cependant ils ne précisent ni le détail du calcul, ni la mise en œuvre de ce test.

2.6.2 Timing Definition Language

Timing Definition Language (*TDL*), le successeur du Giotto, permet d'exécuter concurremment plusieurs modules vus comme des sous-systèmes de tâches dont chacun est autorisé à changer ses modes opérationnels indépendamment des autres modules. Ce changement, contrairement à Giotto, doit être harmonique et ne peut donc se déclencher qu'aux instants qui sont des communs multiples des périodes des tâches du mode.

Exemple 2.10. *Soient les mêmes tâches que dans l'Exemple 2.9 mais reparties dans les modes* m_1, m_2, m'_2 *et les modules* M_1, M_2 *de la façon suivante :* $Modes[M_1] = \{m_1\}$, $Modes[M_2] = \{m_2, m'_2\}$, $\tau[m_1] = \{\tau_1, \tau_2\}$, $\tau[m_2] = \{\tau_3\}$ *et* $\tau[m'_2] = \{\tau'_3\}$. *La période*

de m_1 est égale à 120, celle de m_2 à 40 et celle de m'_2 à 30. Un changement de mode est déclaré possible à la fin des périodes de chacun des modes m_2 et m'_2. La figure ci-dessous illustre le changement du mode m_2 vers le mode m'_2 à l'instant 40. Du point de vue global

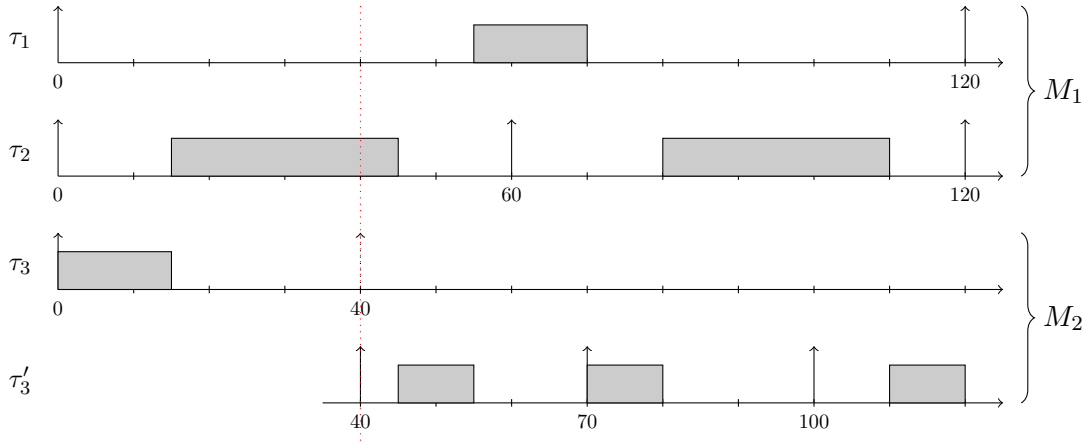


FIGURE 2.33 – Changement de mode dans TDL

du système, le remplacement de la tâche τ_3 par la tâche τ'_3 prend moins de temps pour s'effectuer que dans le cas du même ensemble de tâches exécuté en Giotto dans l'exemple précédent. Les deux exemples sont tirés de [59] où ils sont décrits plus en détail en mettant l'accent sur d'autres aspects du changement de mode dans Giotto *et* TDL.

La question de l'ordonnançabilité de *TDL* sous *EDF* est abordée par Farcas dans sa thèse [59]. En s'appuyant sur les preuves d'ordonnançabilité présentées par Buttazzo [38] et Liu [113], Farcas démontre qu'un ensemble de modules est faisable sous *EDF* si et seulement si la condition suivante est remplie :

$$\sum_{M \in Modules} \max_{m \in Modes[M]} \{U(m)\} \leq 1 \tag{2.53}$$

La somme des taux d'utilisation maximaux des modes sur tous les modules doit être inférieure ou égale à 1.

Dans le même ouvrage, Farcas propose également un test pour des politiques fixes. Il est basé sur l'estimation du pire temps de réponse et repose sur plusieurs principes exposés dans les travaux concernant les protocoles de changement de mode asynchrone (voir section 2.5.2.1). Selon son comportement pendant le changement de mode, la tâche est associée à un des trois groupes : *tâches anciennes*, *tâches inchangées* ou *tâches nouvelles*.

Les temps de réponse des *tâches anciennes*, qui ne sont pas affectées par le changement de mode, sont calculés comme dans les conditions normales.

Dans l'analyse des *tâches inchangées*, les interférences peuvent provenir des autres tâches inchangées, des tâches anciennes et des tâches nouvelles. La Figure 2.34 représente l'estimation de la réponse de la tâche inchangée $\tau_{inchangée}$ en présence de la tâche ancienne $\tau_{ancienne}$ et de la tâche nouvelle $\tau_{nouvelle}$ introduite au changement de mode survenu x unités de temps après l'activation de la tâche inchangée ($\tau_{inchangée}$ s'exécute dans un module M tandis que $\tau_{ancienne}$ et $\tau_{nouvelle}$ dans un module M_{sw}). L'ensemble des valeurs possibles de

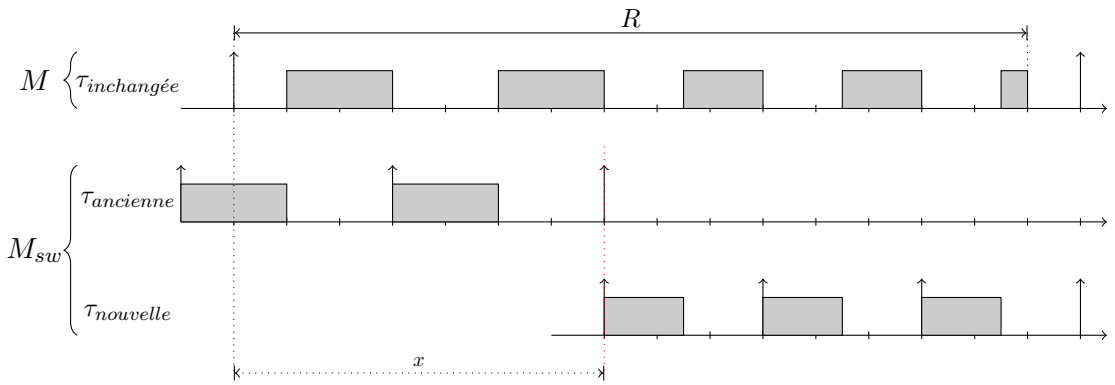


FIGURE 2.34 – Analyse d'une tâche inchangée dans TDL

l'intervalle x peut être déterminé en examinant toutes les périodes des modes qui auraient pu précéder le nouveau mode depuis le démarrage du système. Cette relation sera exploitée dans notre méthode et elle est exposée plus en détail dans la section 3.6. L'analyse du pire temps de réponse de la tâche inchangée peut être surestimée et le test constitue donc une condition suffisante mais pas nécessaire d'ordonnançabilité. Cette surestimation est due à l'impossibilité du calcul exact des interférences provenant des dernières instances des tâches anciennes activées avant la tâche dont le temps de réponse est recherché. Le temps processeur que ces tâches anciennes nécessitent pendant l'exécution de la tâche inchangée ne peut pas être précisément évalué sans connaître le schéma d'ordonnancement exact précédant l'activation de cette tâche. Par exemple, si la tâche ancienne sur la Figure 2.34 avait été préemptée par une autre tâche ancienne, plus prioritaire, avant le démarrage de la tâche inchangée, l'exécution de la tâche ancienne aurait été retardée et se déroulerait à la place de l'exécution de la tâche inchangée. Puisqu'il est impossible d'évaluer analytiquement les parties réelles d'exécution de ces instances, leurs interférences sont bornées supérieurement (à l'image des tâches avortées dans le protocole de Pedro et Burns, voir section 2.5.2.1).

Une *tâche nouvelle* s'exécute en présence d'autres tâches nouvelles et de tâches inchangées. Néanmoins, même si les tâches anciennes ne sont plus actives lors de l'exécution de la tâche nouvelle, elles peuvent avoir un certain impact sur son exécution. En effet, une tâche ancienne peut préempter une tâche inchangée qui, de ce fait, va nécessiter plus de temps de processeur après le changement de mode lorsque toutes les tâches anciennes sont

déjà terminées et que les tâches nouvelles commencent leur exécution. Il est donc nécessaire d'examiner un intervalle de temps qui commence bien avant l'instant du changement de mode. C'est à partir de l'instant, avant le changement de mode, où est activée la tâche inchangée qui a la plus faible priorité parmi toutes les tâches inchangées plus prioritaires que la tâche nouvelle que la fenêtre d'évaluation de son pire temps de réponse est définie. Cette fenêtre est désignée par W_k sur la Figure 2.35 et la notation pour toutes les autres

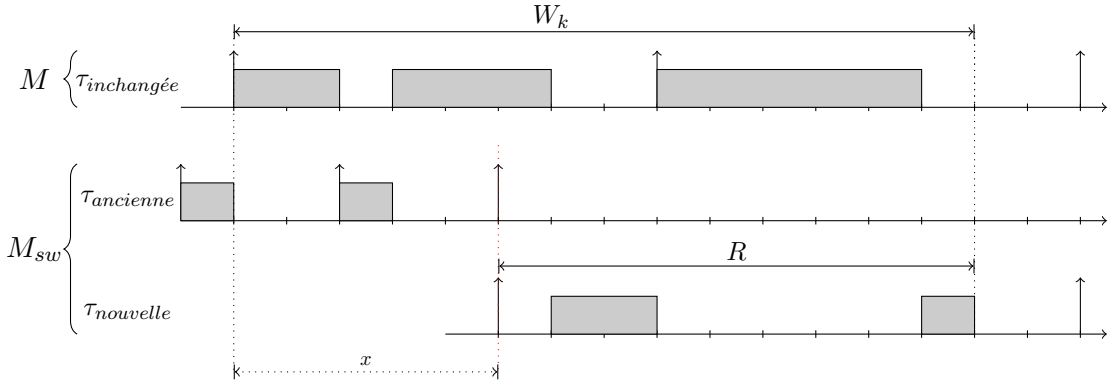


FIGURE 2.35 – Analyse d'une tâche nouvelle dans TDL

tâches reste la même que sur la figure précédente. Quant à l'évaluation des interférences provenant des tâches anciennes, elle peut, comme dans l'analyse des tâches inchangées exposée ci-dessus, être également surestimée.

La procédure pour obtenir les pires temps de réponse de toutes les tâches suit le même déroulement que dans les protocoles asynchrones de changement de mode avec priorités fixes (voir section 2.5.2.1). Ces temps sont calculés de façon itérative comme, par exemple, dans le cas de l'estimation du pire temps de réponse pour *Deadline Monotonic* (voir expression 2.8 et section 2.4.1). Ensuite, ils sont comparés aux périodes afin de vérifier leur ordonnançabilité. Il est à noter que cette méthode permet aussi de vérifier l'ordonnançabilité pour un système de tâches dont les échéances sont contraintes ($D_i \leq T_i$).

2.7 Motivation des travaux pour l'ordonnançabilité d'E-TDL

Ces travaux se situent dans un contexte où le modèle de tâche défini dans les langages dirigés par le temps déjà existants (en particulier *TDL*) a été étendu principalement par l'introduction de déphasages entre les débuts d'exécution de tâches différentes et par le relâchement du lien fixant leurs dates d'échéance à leurs périodes. Ce nouveau modèle, fondant une extension *E-TDL* de *TDL*, est défini et décrit en section 1.5.2. La question est de savoir s'il existe une technique qui permette de vérifier l'ordonnançabilité d'un système composé sur la base de ce nouveau modèle.

Le test d'ordonnançabilité sous des politiques à priorité fixe pour *TDL* discuté auparavant permet de tenir compte de dates d'échéance inférieures aux périodes des tâches. Néanmoins, dans *E-TDL*, des tâches ayant des offsets différents de zéro ne peuvent jamais démarrer en même temps. L'application du test proposé par Farcas [59] dans ce cas particulier peut être donc pessimiste. Le choix d'offsets peut impliquer des déphasages entre activations de tâches. Ces déphasages ne sont pas pris en compte par le test et le départ simultané de toutes les tâches est toujours considéré. Pour réduire le pessimisme de cette approche, on peut envisager d'adopter des techniques d'analyse d'algorithmes d'ordonnancement à priorité fixe avec offsets [170] ou de définir toutes les configurations dans lesquelles les tâches peuvent effectivement démarrer.

Grâce à l'équivalence entre le modèle de tâche dans *TDL* et le modèle classique de Liu et Layland ($\tau_i = (C_i, T_i)$), le problème de l'ordonnançabilité de *TDL* sous *EDF* peut facilement être résolu avec le test du taux d'utilisation. Ce n'est plus le cas d'*E-TDL* car, comme vu en section 2.4.2, si les échéances des tâches ne sont pas égales à leurs périodes, chaque intervalle du cycle d'ordonnancement doit être examiné avec la *fonction de demande*.

Au travers de l'exemple suivant nous montrons comment les différentes analyses d'ordonnancement exposées dans les points précédents s'appliquent au problème d'ordonnançabilité d'*E-TDL* sous *EDF*. Cela permet d'identifier et de repérer les principales difficultés de l'analyse d'ordonnançabilité pour *E-TDL* mais aussi de s'interroger plus généralement sur l'analyse des systèmes dirigés par le temps.

Exemple 2.11. *Soit un système E-TDL composé de deux modules* $Modules = \{M_1, M_2\}$. *Le premier module,* M_1*, peut s'exécuter dans un des deux modes :* $Modes[M_1] = \{m_1, m_1'\}$. *Les périodes des deux modes* $T[m_1]$ *et* $T[m_1']$ *sont toutes deux égales à* 4. *A la fin de ces périodes, si la condition de changement de mode est vérifiée, l'exécution de l'autre mode peut se déclencher, sinon, le mode courant redémarre. Le deuxième module,* M_2*, s'exécute dans un seul mode,* $Modes[M_2] = \{m_2\}$*, dont la période* $T[m_2]$ *est égale à* 8. *Chaque mode, du module* M_1 *et du module* M_2*, exécute une seule tâche dont les paramètres* (Φ, C, TEL, T) *sont :* $\tau[m_1] = \{\tau_1 = (0, 3, 3, 4)\}$, $\tau[m_1'] = \{\tau_1' = (3, 1, 1, 4)\}$, $\tau[m_2] = \{\tau_2 = (0, 2, 8, 8)\}$. *La Figure 2.36 illustre l'exécution de ce système. Cette trace d'exécution comporte plusieurs schémas d'exécution, dont ceux qui produisent la plus grande charge et permet donc de conclure que le système est ordonnançable. On remarque également que l'activation du mode* m_2 *coïncide dans tous les cas possibles avec les activations des modes* m_1 *ou* m_1'.

Pour prouver théoriquement l'ordonnançabilité du système de l'Exemple 2.11 il pourrait être fait un appel à l'une des méthodes d'analyse compositionnelle présentées dans la section 2.5.4. Néanmoins, aucune d'elles ne considère explicitement un modèle de tâche doté, comme dans *E-TDL*, de quatre paramètres (offset, pire temps d'exécution, échéance et période) se bornant au plus à un modèle de tâche sporadique. Parmi ces méthodes, *Real-Time Calculus* permet d'exprimer [177] des schémas d'activation de tâches non strictement périodiques, voire arbitraires. Chaque composant est caractérisé par les ressources maximales de processeur demandées pendant une certaine durée. Cette approche ciblant les systèmes

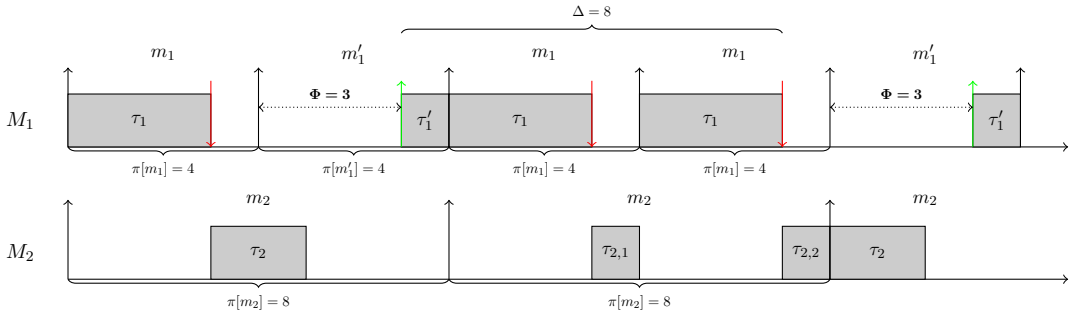


FIGURE 2.36 – Une trace d'exécution de deux modules E-TDL

événementiels fait l'hypothèse d'un éventuel synchronisme de début de toutes les activités au sein des différents modules. Cela permet d'évaluer la demande globale produite par tous les composants du système comme la somme des demandes de chacun d'eux sur le même intervalle (voir Formule 2.46). Autrement dit, dans un système évènementiel il existe un instant unique pour tous les composants de ce système à partir duquel chacun d'eux produit la plus grande charge. L'exemple précédent permet de montrer que ce type de composition est aussi applicable dans le contexte des systèmes dirigés par le temps. Cependant, il se peut que certaines activités, dans deux modules différents, ne puissent se produire au même instant ce qui risque de conduire à une surestimation de la valeur de la charge totale du système.

Exemple 2.12. *Dans la trace d'exécution sur la Figure 2.36 on considère les intervalles de temps Δ dont la durée est 8. Dans le module M_1, la plus grande charge exécutée dans un intervalle de cette durée est celle indiquée sur la dite figure avec une accolade et est égale à 7. Dans le module M_2, la plus grande charge est associée à l'exécution entière du mode m_2 et s'élève à 2. La somme de ces deux charges est égale à 9 et dépasse la durée de l'intervalle ($9 > 8$). Cela contredit la condition d'ordonnançabilité d'EDF (voir Condition 2.17 ou 2.46) et ne permet pas de garantir la faisabilité du système. Pourtant, le système est ordonnançable. Cette contradiction vient de ce que les intervalles dont les charges ont été additionnées ne commencent jamais au même instant, conduisant à une surestimation de la charge totale du système.*

Dans la section 1.2 les systèmes à déclenchement événementiel et temporel ont été comparés. Les systèmes à déclenchement temporels observent l'état du processus contrôlé uniquement à des instants bien prédéfinis et ne déclenchent les activités appropriées qu'à ces instants-là. Les instants d'activation des tâches, des changements de mode potentiels, des lectures et des écritures sont tous précisément déclarés. Il est donc possible de déterminer si deux activités différentes peuvent se produire au même instant ou pas. L'analyse compositionnelle de tels systèmes, du point de vue de l'ordonnançabilité, doit être en mesure de situer précisément les débuts des intervalles étudiés dans les différents modules (composants)

du système et doit comparer seulement les intervalles dont les débuts peuvent être concomitants [99].

La proposition d'une méthode d'analyse de ce type pour les systèmes dirigés par le temps s'inscrit naturellement dans le cadre des objectifs de cette thèse et est développée dans les chapitres suivants de ce mémoire. De nombreuses approches présentées au sein de ce chapitre sont utilisées afin de concourir à formuler un test d'ordonnançabilité pour *E-TDL* sous *EDF*.

Chapitre 3

Ordonnançabilité pour E-TDL

Ce chapitre traite du problème de l'ordonnançabilité d'un système composé de plusieurs modules E-TDL. Les conditions d'ordonnançabilité proposées s'appliquent au cadre particulier de la politique d'ordonnancement *EDF* (*Earliset Deadline First*) sur *monoprocesseur*.

L'approche consiste d'abord à décrire et formaliser les évolutions temporelles d'un module qui sont imposées par la sémantique du langage. Identifier les différents schémas d'activation des tâches déclenchées au sein d'un module permet d'évaluer le temps de processeur nécessaire à leur exécution. Les conditions d'ordonnançabilité pour EDF proposées par Baruah [24] établissent qu'un système de tâches est faisable s'il n'existe aucun intervalle sur lequel le temps d'exécution requis par les tâches excède le temps de processeur disponible. L'évaluation pratique de cette condition nécessite de fixer une borne sur la longueur des intervalles qui doivent être vérifiés. Cette borne une fois fixée, il devient possible de formuler des conditions pour le test d'ordonnançabilité exact d'un système composé d'un seul module ainsi que pour le test approché d'un système composé de plusieurs modules. Ce dernier est basé sur les tests pour les systèmes à déclenchement événementiel. Donc, pour les raisons exposées en section 2.7, parce qu'il somme les demandes de temps de processeur pour les différents modules comme si celles-si étaient simultanées, ce test surestime la demande cumulative. D'où l'importance d'étudier finement le synchronisme entre les activités invoquées au sein des différents modules et les relations qui conditionnent leur simultanéité. Savoir de quelle manière se positionnent les exécutions dans les modules permet de ne considérer uniquement que les demandes de temps de processeur faites réellement simultanément et donne une réponse exacte à la question de l'ordonnançabilité du système.

3.1 Introduction

Une tâche périodique $\tau_i = (\Phi_i, C_i, TEL_i, T_i)$ est caractérisée dans *E-TDL* par un offset Φ_i, son pire temps d'exécution C_i, un Temps d'Exécution Logique TEL_i et une période T_i. L'offset Φ_i doit être inférieur à la période T_i :

$$0 \leq \Phi_i < T_i \tag{3.1}$$

Le Temps d'Exécution Logique TEL_i doit être suffisamment étendu pour que la tâche τ_i puisse s'exécuter dans ses limites :

$$0 < C_i \leq TEL_i \leq T_i - \Phi_i \tag{3.2}$$

La sémantique du langage interdit ainsi que la $j^{ème}$ instance ($j \in \mathbb{N}_+$) de la tâche τ_i, activée dans l'intervalle $[(j-1)T_i, jT_i]$, se termine après l'instant jT_i. Dès lors, les dates d'activation $r_{i,j}$ et d'échéance $d_{i,j}$ de la tâche périodique τ_i sont données par :

$$r_{i,j} = \Phi_i + (j-1) \cdot T_i \ \leq \ j \cdot T_i \tag{3.3}$$

$$d_{i,j} = \Phi_i + TEL_i + (j-1) \cdot T_i \ \leq \ j \cdot T_i \tag{3.4}$$

La Figure 3.1 représente une tâche périodique (telle qu'introduite en section 2.1) dont l'exécution se conforme à la sémantique imposée par *E-TDL*.

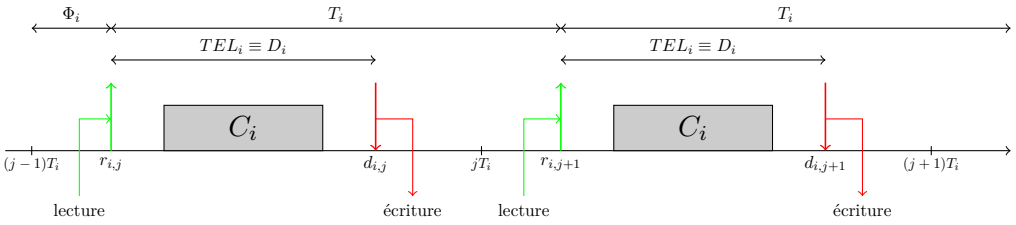


FIGURE 3.1 – Tâche conforme à la sémantique E-TDL

On ajoute aux contraintes précédentes, les hypothèses suivantes :

(H1) Les tâches sont périodiques et leurs périodes sont constantes dans chaque mode.

(H2) Une nouvelle instance de tâche ne peut être invoquée que si l'instance courante est terminée.

(H3) Les tâches sont indépendantes ; l'invocation d'une tâche ne dépend pas de l'invocation ou de la terminaison d'autres tâches.

(H4) Les tâches ne demandent pas d'autres ressources que le processeur.

(H5) Les tâches s'exécutent sur un seul processeur. La vitesse du processeur est constante.

(H6) A tout instant, une seule tâche peut s'exécuter sur le processeur.

(H7) Une tâche peut être préemptée à tout moment. La durée de la préemption est considérée nulle.

(H8) La durée de lecture des entrées d'une tâche et la durée d'écriture de ses sorties sont considérées nulles.

(H9) Tous les modes sont atteignables à partir du mode initial.

(H10) Il n'est fait aucune hypothèse sur les séquences d'activation pouvant conduire au changement de mode d'un module

3.2 Evolution temporelle d'un module

La connaissance de tous les comportements possibles d'un système permet d'évaluer la charge de calcul qu'ils génèrent. Il est ainsi primordial de connaître précisément ces comportements pour ensuite les quantifier en termes de temps processeur nécessaire à leur exécution.

3.2.1 Graphe de changement de mode

En changeant de modes opérationnels chaque module adapte son fonctionnement et ses activités à l'état du processus contrôlé. L'évolution de ce processus peut être représentée par une séquence d'états et de transitions. Un module réalise également une séquence d'exécution de ses différents modes. Le prochain mode d'exécution d'un module est pris parmi les modes qui sont les cibles des transitions définies depuis le mode courant. Ces transitions ne peuvent se produire qu'aux instants d'évaluation des conditions de changement de mode. La structure d'un graphe orienté paraît ainsi la mieux adaptée pour décrire l'évolution temporelle d'un module. Farcas introduit un tel graphe pour *TDL* [59] et l'appelle *graphe de changement de mode* (*mode-transition graph*). Puisque les mécanismes du changement de mode sont identiques dans *TDL* et *E-TDL*, ce graphe de changement de mode peut être utilisé sans subir aucune modification. Les modes sont représentés par les nœuds du graphe et les transitions par les arcs étiquetés avec les périodes auxquelles leurs conditions sont évaluées.

Définition 3.1 (**Graphe de changement de mode**). *Le graphe de changement de mode d'un module $M_j \in Modules$, décrit par un texte E-TDL, est défini par un ensemble de transitions. Une transition est la donnée d'un triplet $(m_k, m_{k+1}, T_{sw}(m_k, m_{k+1}))$ où m_k et m_{k+1} sont des modes de M_j tels qu'un commutateur de changement de mode de m_k à m_{k+1} est déclaré dans le module M_j et dont la condition est périodiquement évaluée dans m_k toutes les $T_{sw}(m_k, m_{k+1})$ unités de temps. Le redémarrage du mode m_k à la fin de sa période $T[m_k]$ est aussi considéré comme une transition $(m_k, m_k, T[m_k])$. Formellement,*

l'ensemble des transitions est donné par :

$$\{(m_k, m_{k+1}, T_{sw}(m_k, m_{k+1})) \mid m_k, m_{k+1} \in Modes[M_j], \\ \exists \delta_k : m_{k+1} \in prochains\_modes(m_k, \delta_k)\} \tag{3.5}$$

Exemple 3.1. *La Figure 4.4 représente le graphe de changement de mode d'un module composé des modes m_a, m_b et m_c. Dans le mode initial m_a, les conditions de changement de mode vers les modes m_b et m_c sont périodiquement évaluées, respectivement, toutes les $T_{sw}(m_a, m_b)$ et toutes les $T_{sw}(m_a, m_c)$ unités de temps. Dans le mode m_b, la condition de changement de mode est périodiquement évaluée toutes les $T_{sw}(m_b, m_c)$ unités de temps. Aucun autre changement de mode n'est possible.*

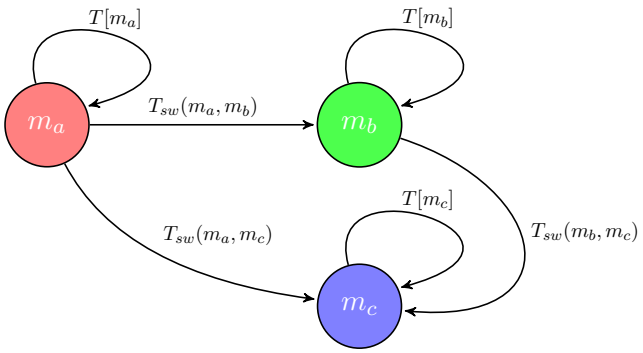


FIGURE 3.2 – Graphe de changement de mode

Le comportement temporel d'un module peut être décrit par un *chemin* qui traverse le graphe de changement de modes et visite les différents modes. Le temps passé dans un mode est un multiple de la période d'évaluation du changement de ce mode vers le suivant.

Définition 3.2 (Chemin dans le graphe de changement de mode). *Un chemin w dans un graphe de changement de mode est une séquence de n paires $((m_1, \mu_1), \ldots, (m_n, \mu_n))$ telles que $m_1, \ldots, m_n$ sont des modes consécutifs dans le graphe de changement de mode et μ_k représente :*

- *pour $k < n$, $\mu_k \in \mathbb{N}_+$: le nombre de fois que la condition pour rejoindre le mode m_{k+1} a été évaluée dans m_k avant que l'exécution dans le mode m_{k+1} ne soit démarrée,*
- *pour $k = n$, $\mu_k \in \mathbb{N}_0$: le nombre de périodes entières $T[m_k]$ passées dans le mode m_k.*

Exemple 3.2. *Soit un module décrit en E-TDL dont le graphe de changement de mode est celui de l'Exemple 3.1. Le chemin $w = ((m_a, 3), (m_b, 5), (m_c, 1))$ correspond à l'exécution de mode m_a pendant trois périodes $T_{sw}(m_a, m_b)$ suivie de l'exécution de mode m_b pendant cinq périodes $T_{sw}(m_b, m_c)$ après lesquelles le mode m_c s'exécute pour une période $T[m_c]$.*

Les deux définitions suivantes permettent de désigner le premier et le dernier mode dans un chemin traversant un graphe de changement de mode.

Définition 3.3 (**Début de chemin**). *Soit un chemin* $w = ((m_1, \mu_1), \ldots, (m_n, \mu_n))$. *Par* $head(w)$, *on désigne le premier mode exécuté dans ce chemin :* $head(w) = m_1$.

Définition 3.4 (**Fin de chemin**). *Soit un chemin* $w = ((m_1, \mu_1), \ldots, (m_n, \mu_n))$. *Par* $tail(w)$ *on désigne le dernier mode exécuté dans ce chemin :* $tail(w) = m_n$.

Définition 3.5 (**Longueur de chemin**). *Soit un chemin* $w = ((m_1, \mu_1), \ldots, (m_n, \mu_n))$. *La longueur* $|w|$ *du chemin* w *est donnée par :*

$$|w| = \sum_{k=1}^{n} \mu_k \cdot T_{sw}(m_k, m_{k+1}) \tag{3.6}$$

où le terme $T_{sw}(m_n, m_{n+1})$ *peut être remplacé par* $T[m_n]$ *pour* $k = n$.

3.2.2 Trace d'exécution d'un module E-TDL

Un chemin démarre à l'instant δ_1, passé dans le premier mode m_1, multiple de la période d'évaluation des conditions de changement de mode vers le second mode (δ_1 mod $T_{sw}(m_1, m_2) = 0$). Il finit au début du dernier mode m_n de ce chemin ($\delta_n = 0$). Pour pouvoir décrire le comportement d'un module dans tout intervalle temporel, on introduit la notion de *trace d'exécution d'un module*. La Figure 3.3 illustre une trace observée dans un intervalle de temps $[t_s, t_f]$ tel que t_s coïncide avec l'état (m_s, δ_s) et l'instant t_f avec l'état (m_f, δ_f). Une trace encapsule un chemin. Ce chemin démarre après t_s à l'instant de redémarrage du mode m_s ou du premier changement de mode depuis m_s. Il se termine à l'instant de la dernière terminaison de mode précédant t_f.

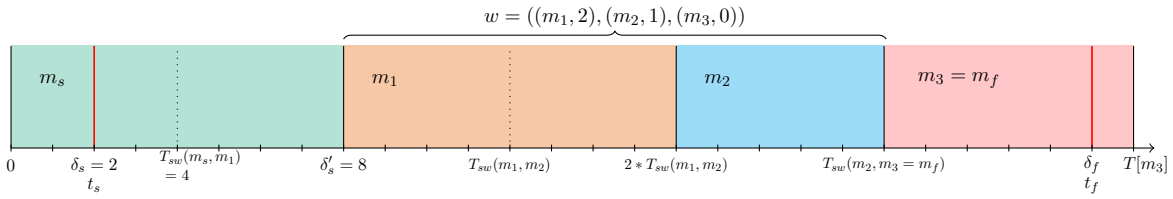


FIGURE 3.3 – Trace d'exécution d'un module

Définition 3.6 (**Trace d'exécution d'un module**). *Une trace d'exécution d'un module* M_j *dans l'intervalle de temps* $[t_s, t_f]$ *est notée* $\sigma_{M_j} = (m_s, \delta_s, \delta'_s, w, m_f, \delta_f)$ *où :*

— $m_s \in Modes[M_j]$ *est le mode dont l'exécution précède l'exécution du chemin* w,

— $\delta_s : 0 < \delta_s \leq T[m_s]$ *est le temps de mode* m_s *correspondant à l'instant* t_s,

— $\delta'_s : \delta_s \leq \delta'_s \leq T[m_s]$ *est le temps de mode* m_s *correspondant à l'instant de démarrage de* w,

— w *est un chemin dans le graphe de changement de mode de module* M_j,

— $m_f \in Modes[M_j]$ *est le mode dont l'exécution fait suite à l'exécution du chemin* w,

— $\delta_f : 0 \leq \delta_f < T[m_f]$ *est le temps de mode* m_f *correspondant à l'instant* t_f.

Les conditions suivantes doivent être remplies :

— $head(w) \in prochains\_modes(m_s, \delta'_s)$,

— *si* $tail(w) \neq m_f$, *alors* $m_f \in prochains\_modes(tail(w), T[tail(w)])$ *et le mode* $tail(w)$ *est exécuté au moins une fois à la fin du chemin* w,

— $t_f - t_s = \delta'_s - \delta_s + |w| + \delta_f$.

Les intervalles d'exécution de courte durée, pendant lesquels le mode m_s ne s'exécute plus que pendant une période, peuvent être représentés par $\sigma_{M_j} = (m_s, \delta_s, \delta'_s, \emptyset, \emptyset, 0)$.

La longueur d'une trace d'exécution σ_{M_j} s'exprime par :

$$|\sigma_{M_j}| = \delta'_s - \delta_s + |w| + \delta_f \tag{3.7}$$

On désigne par $start(\sigma_{M_j}) = (m_s, \delta_s)$, le mode et le temps de ce mode, du démarrage d'une trace σ_{M_j}.

3.3 Caractéristique de la charge du processeur produite par un module

On peut associer à une trace d'exécution d'un module le temps de processeur nécessaire pour exécuter les tâches qu'elles déclenchent. La section précédente établissait qu'une trace est composée de trois intervalles : l'exécution avant le chemin encapsulée par la trace, le chemin et l'exécution après le chemin. Les exécutions avant et après le chemin encapsulé s'étendent au sein d'un mode et ne durent pas au delà de la période du mode courant. L'exécution d'un chemin comporte plusieurs exécutions entières de modes. Une tâche dont la date d'activation se trouve dans l'un de ces intervalles a donc toujours sa date d'échéance située dans le même intervalle. La demande de temps de processeur associée à l'exécution d'une trace est la somme des demandes associées à chacun de ces trois intervalles.

On quantifie d'abord la demande de temps de processeur dans les limites d'une période de mode. Cette demande peut être associée à l'exécution dans l'intervalle qui précède l'exécution du chemin ou à celui qui lui succède. On s'appuie sur les formules 2.13 et 2.12 qui expriment la demande de temps de processeur des tâches à échéance avant requête.

Définition 3.7 (**Fonction de demande dans un mode**). *Soit δ_k et δ'_k des temps de mode m_k tels que : $0 \leq \delta_k < \delta'_k \leq T[m_k]$. La fonction de demande $df(m_k, \delta_k, \delta'_k)$ dans le mode m_k est la durée d'exécution cumulée des instances de tâches dont les dates d'activation et les échéances se situent entre les instants de mode δ_k et δ'_k dans le mode m_k :*

$$df(m_k, \delta_k, \delta'_k) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau[m_k]} \max\left(0, \left(\left\lfloor \frac{\delta'_k - \Phi_i - TEL_i}{T_i} \right\rfloor - \left\lceil \frac{\delta_k - \Phi_i}{T_i} \right\rceil + 1\right) C_i\right) \tag{3.8}$$

Un chemin exécute une séquence d'hyperpériodes de différents modes. La charge du processeur produite pendant l'exécution complète d'une hyperpériode des tâches d'un mode est proportionnelle au taux d'utilisation des tâches de ce mode (voir Formule 2.2).

Définition 3.8 (**Fonction de demande d'un chemin**). *La fonction de demande d'un chemin w, $df(w)$, est la durée d'exécution de toutes les tâches exécutées dans ce chemin :*

$$df(w) \stackrel{\text{def}}{=} \sum_{(m_k, \mu_k) \in w} \mu_k \cdot T_{sw}(m_k, m_{k+1}) \cdot U(m_k) \tag{3.9}$$

où $U(m_k)$ est le taux d'utilisation de processeur pour le mode m_k :

$$U(m_k) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau[m_k]} \frac{C_i}{T_i}. \tag{3.10}$$

Les deux définitions ci-dessus permettent d'exprimer la demande des ressources de processeur associée à l'exécution d'une trace d'un module d'E-TDL :

Définition 3.9 (**Fonction de demande d'une trace de module**). *La fonction $df(\sigma_{M_j})$ est le temps processeur nécessaire à l'exécution d'une trace $\sigma_{M_j} = (m_s, \delta_s, \delta'_s, w, m_f, \delta_f)$ d'un module.*

$$df(\sigma_{M_j}) \stackrel{\text{def}}{=} df(m_s, \delta_s, \delta'_s) + df(w) + df(m_f, 0, \delta_f) \tag{3.11}$$

Chaque trace décrit une évolution particulière d'un module dans le temps. Les changements de mode ne sont pas connus d'avance. Ils sont déclenchés à la suite d'un changement survenu dans le système ou dans son environnement. Tous les changements de mode spécifiés dans une description E-TDL sont potentiellement possibles. Donc, le module, à partir d'un état donné, peut suivre toutes les traces d'exécutions possibles à partir de cet état.

Exemple 3.3. *Soit un module M_1 qui peut s'exécuter dans les modes $Modes[M_1] = \{m_1, m_2\}$. Ces deux modes ont la même période $T[m_1] = T[m_2] = T$. A la fin d'une période du mode m_1, le module M_1 peut changer de mode et passe de m_1 à m_2 si le relevé d'un capteur dépasse une certaine valeur de seuil. Dans l'intervalle $[T, 2T]$, on peut donc observer les traces d'exécution dans l'ensemble suivant :*

$$\{(m_1, T, T, w, m_f, 0) \mid w \in \{(m_1, 1), (m_2, 1)\}, m_f = tail(w)\}$$

Dans le cadre de l'analyse d'ordonnançabilité on cherche à déterminer la plus grande charge possible du processeur dans un intervalle de temps donné. Si l'on observe un module pendant un intervalle Δ de temps qui commence dans un état donné, on considère seulement, parmi toutes les traces qui sont exécutables à partir de cet état, celle dont la valeur de fonction de demande est la plus importante sur l'intervalle Δ.

Définition 3.10 (**Fonction de demande maximale dans un module**). *La fonction de demande maximale* $maxdf_{M_j}(m_s, \delta_s, \Delta)$ *désigne la plus grande valeur de la fonction de demande parmi les traces du module* M_j *qui commencent lorsque ce module exécute le mode* $m_s \in Modes[M_j]$ *à l'instant de mode* δ_s *et sont de durée* Δ.

$$maxdf_{M_j}(m_s, \delta_s, \Delta) \stackrel{\text{def}}{=} \max_{\sigma_{M_j}} \{df(\sigma_{M_j}) \mid start(\sigma_{M_j}) = (m_s, \delta_s), |\sigma_{M_j}| = \Delta\} \tag{3.12}$$

La définition suivante permet d'évaluer la demande de temps de processeur indépendamment de l'état du module et seulement en fonction de la durée de l'intervalle. On observe dans le module toutes les traces possibles de durée Δ quelle que soit l'origine de cette observation. La définition ci-dessous est dérivée de la notion de fonction $dbf(\Delta)$ donnée par la formule 2.16.

Définition 3.11 (**Borne de fonction de demande maximale dans un module**). *Pour un module* M_j*, la borne de fonction de demande maximale* $mdbf_{M_j}(\Delta)$ *est égale à la plus grande valeur de la fonction de demande maximale associée à une des traces d'exécution de longueur* $\Delta > 0$.

$$mdbf_{M_j}(\Delta) \stackrel{\text{def}}{=} \max_{(m_s, \delta_s)} \{maxdf_{M_j}(m_s, \delta_s, \Delta)\} \tag{3.13}$$

3.4 Condition d'ordonnançabilité pour un module

E-TDL n'impose aucune restriction sur le nombre des modules. Dans le cas particulier où le système est composé d'un seul module, l'analyse d'ordonnançabilité devient plus simple. Les notations introduites dans la section précédente permettent alors d'évaluer la charge de calcul pouvant être générée par un module. On combine ces méthodes avec la condition d'ordonnançabilité proposée par Baruah pour les tâches à échéances avant requêtes [24], exposée dans la section 2.4.2 de ce mémoire, afin d'obtenir une condition nécessaire et suffisante d'ordonnançabilité d'un système qui exécute un seul module.

On démontre d'abord qu'aucune tâche ne s'exécute à un instant multiple de l'hyperpériode du mode courant.

Lemme 3.1. *Soit un système défini par un ensemble* $Modules$ *de modules. Si le système est ordonnançable, les tâches exécutées par tout module* $M \in Modules$*, dans tout mode* $m \in Modes[M]$*, sont terminées lorsque le module passe dans l'état* (m, δ) *avec* $\delta \bmod H[m] = 0$.

Démonstration. Considérons un module $M \in Modules$ et un ensemble de n tâches $\tau[m] = \{\tau_1, \ldots, \tau_n\}$ dans un mode $m \in Modes[M]$. Pour chaque i tel que $1 \leq i \leq n$ on pose $k_i = H[m]/T_i = ppcm\{T_1, \ldots, T_n\}/T_i$. La relation 3.4 précise que la $k_i^{ème}$ instance de la tâche $\tau_i \in \tau[m]$ doit être finie au plus tard à l'instant $d_{i,k_i} \leq k_i T_i = H[m]$. Si aucun changement de mode n'intervient à l'instant $H[m]$, d'après la relation 3.3, l'instant de la prochaine requête est égal à $r_{i,k_i+1} = \Phi_i + k_i T_i = \Phi_i + H[m] \geq H[m]$. Ceci implique qu'aucune des tâches de l'ensemble $\tau[m]$ ne s'exécute dans l'intervalle $\cap_{1 \leq i \leq n}[d_{i,k_i}, r_{i,k_{i+1}}]$. Par ailleurs, $H[m] \in \cap_{1 \leq i \leq n}[d_{i,k_i}, r_{i,k_{i+1}}]$. Donc le processeur n'exécute aucune tâche de mode m à l'instant de mode égal à $H[m]$. Si le module M continue l'exécution de mode m après cet instant, les mêmes tâches sont activées à l'instant de mode δ ainsi qu'à l'instant de mode $\delta + H[m]$ pour δ $0 \leq \delta \leq H[m]$. Le processeur n'exécute donc aucune tâche de mode m à tout instant de mode multiple de son hyperpériode $H[m]$. □

Si le système n'a qu'un seul module, le processeur est inoccupé lorsque l'exécution d'un de ses modes arrive à l'hyperpériode. Les périodes d'activité du processeur s'étalent donc entre ces instants et c'est uniquement durant ces périodes que le dépassement d'une échéance peut avoir lieu. Selon le Lemme 3.2 l'ordonnançabilité d'un tel système est impliquée par l'ordonnançabilité de chacun de ses modes analysé séparément. Farcas a obtenu le même résultat pour *TDL* classique [59].

Lemme 3.2. *Soit un système défini par un seul module. Le système est ordonnançable sur un processeur si et seulement si chacun de ses modes est ordonnançable.*

Démonstration. Un module exécute un mode à la fois. Chaque mode s'exécute sur un processeur depuis le début jusqu'à la fin de sa période ou jusqu'à un instant de changement de mode. Les instants de changement de mode surviennent à des instants multiples d'hyperpériodes de mode. D'après le Lemme 3.1, si le système est faisable, aucune tâche ne s'exécute à un instant de mode multiple de l'hyperpériode du mode. Par conséquent, si le processeur est attribué à un seul module le processeur est inoccupé à tout instant multiple de l'hyperpériode du mode. Un dépassement d'échéance ne peut se produire que dans les périodes d'activité du processeur. Ces périodes s'étendent, dans le pire des cas, entre deux instants consécutifs d'hyperpériode d'un mode. Seules les tâches d'un mode peuvent s'exécuter durant ces intervalles de temps. Les modes étant ordonnançables, le système l'est donc aussi. □

D'après le Lemme 3.2 l'analyse d'ordonnançabilité d'un système composé d'un seul module peut être décomposée en l'analyse de chacun de ses modes. Le Lemme 3.1 permet de borner l'intervalle de recherche à la durée d'une hyperpériode de mode. Il reste donc à vérifier l'ordonnançabilité de chaque mode dans les limites de son hyperpériode en appliquant le test proposé par Baruah [24] pour les tâches à échéance avant requête.

Théorème 1. *Soit un système défini par un seul module : M. Le système est ordonnançable sur un processeur sous EDF si et seulement si pour tout mode $m \in Modes[M]$:*

$$1. \quad U(m) \leq 1, \ \ et \tag{3.14}$$

$$2. \quad df(m, \delta_1, \delta_2) \leq \delta_2 - \delta_1 \ \ pour \ \ tous \ \ \delta_1 \ \ et \ \ \delta_2 \ \ tels \ \ que \ \ 0 \leq \delta_1 < \delta_2 \leq H[m] \tag{3.15}$$

Démonstration. On montre que le problème d'ordonnançabilité d'un système défini par un seul module est équivalant au problème d'ordonnançabilité pour les tâches à échéances avant requêtes [24]. Grâce au Lemme 3.2, l'ordonnançabilité du système composé d'un seul module est prouvée si et seulement si tous les modes de ce module sont ordonnançables. Le Lemme 3.1 indique que les périodes d'activité du processeur pour un tel système se produisent dans des intervalles de temps situés entre hyperpériodes de mode. Un dépassement d'échéance ne peut être observé qu'à la suite d'une période continue d'activité de processeur. Les mêmes tâches sont activées aux instants de mode séparés exactement par des multiples de l'hyperpériode du mode dans lequel elles s'exécutent et les états d'exécution de ces tâches à ces instants sont aussi les mêmes. Une tâche de mode m dépasse donc une de ses échéances seulement si elle en dépasse une dans l'intervalle $[\delta_1, \delta_2] \subset [0, H[m]]$. Les schémas d'ordonnancement dans l'intervalle $[0, H[m]]$ sont périodiques et seulement les tâches d'un même mode s'y exécutent. Il est donc possible de vérifier la faisabilité de cet ensemble dans cet intervalle en appliquant la condition d'ordonnançabilité pour les tâches à échéances avant requêtes proposée par Baruah [24], dont la preuve se trouve dans la section 2.4.2. □

Exemple 3.4. *Considérons un système à un seul module, $Modules = \{M\}$, et deux modes $Modes[M] = \{m_1, m_2\}$. La période du mode m_1, $T[m_1]$, est égale à 12, et la période du mode m_2, $T[m_2]$, à 8. Les changements de mode de m_1 à m_2 et de m_2 à m_1 peuvent se déclencher en fin des périodes de modes. Dans le premier mode s'exécutent les tâches : $\tau[m_1] = \{\tau_{11} = (0, 2, 2, 12), \tau_{12} = (0, 3, 6, 6)\}$ et dans le second les tâches : $\tau[m_2] = \{\tau_{21} = (0, 3, 4, 4), \tau_{22} = (2, 1, 3, 8)\}$. La Figure 3.4 illustre un schéma d'exécution de ce système.*

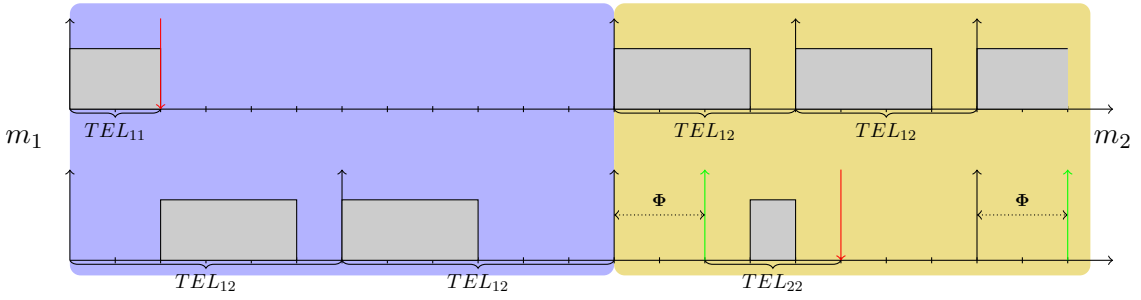


FIGURE 3.4 – Schéma d'exécution du système de l'Exemple 3.4.

L'ordonnançabilité de ce système est vérifiée en utilisant le résultat du Théorème 1. Le système est ordonnançable si chacun de ses deux modes, m_1 et m_2, est ordonnançable. Le taux d'utilisation de processeur pour le mode m_1 s'élève à 67%, et la première condition du théorème (3.14) est vérifiée. Il reste maintenant à calculer la fonction de demande (3.8) pour les intervalles dans le mode m_1. On se borne aux intervalles dont le début coïncide avec une date de démarrage et la fin avec une date d'échéance d'une tâche. La valeur de la fonction de demande dans ces intervalles est donnée par :

- $df(m_1, 0, 2) = 2 \leq 2$,
- $df(m_1, 0, 6) = 5 \leq 6$,
- $df(m_1, 0, 12) = 8 \leq 12$,
- $df(m_1, 6, 12) = 3 \leq 6$.

La seconde condition (3.15) du Théorème 1 est vérifiée pour le mode m_1. Le même procédé est répété pour le mode m_2. Son taux d'utilisation ($87,5\%$) remplit la première condition du théorème (3.14). Les valeurs de la fonction de demande sont calculées comme suit :

- $df(m_2, 0, 4) = 3 \leq 4$,
- $df(m_2, 0, 8) = 7 \leq 8$,
- $df(m_2, 2, 5) = 1 \leq 3$,
- $df(m_2, 4, 8) = 3 \leq 4$.

La condition (3.15) du Théorème 1 est donc vérifiée. Les modes m_1 et m_2 sont ordonnançables, ce qui implique l'ordonnançabilité du système entier.

3.5 Condition suffisante d'ordonnançabilité pour plusieurs modules

La section 2.7 a montré que les méthodes basées sur la demande de processeur pour l'analyse d'ordonnançabilité des systèmes à déclenchement événementiel sous *EDF* peuvent fournir une condition suffisante, mais pas nécessaire, dans le cas des systèmes modélisés avec *E-TDL*. Avec les fonctions de demande pour *E-TDL* il est possible de quantifier le temps maximal de processeur requis par les tâches d'un module dans tous les intervalles. Cela permet de caractériser un système *E-TDL* faisable. Mais, pour que ce test devienne praticable, il est nécessaire de borner la longueur des intervalles à vérifier. A l'aide de ces deux éléments, une condition suffisante pour l'ordonnançabilité d'un système défini par un ensemble de modules *E-TDL* peut être formulée.

3.5.1 Caractéristique d'un système E-TDL ordonnançable sous EDF

La plupart des tests d'ordonnançabilité pour l'analyse d'un ensemble de tâches à échéance avant requête sous EDF sont basés sur l'observation selon laquelle le système n'est

pas faisable s'il existe un intervalle de temps tel que le temps de processeur requis par les tâches qui s'y exécutent dépasse le temps de processeur disponible dans cet intervalle [24]. La fonction $mdbf_M(\Delta)$ donne le plus grand temps d'exécution demandé par les tâches du mode M dans un intervalle de durée Δ. Tous les modules de l'ensemble $Modules$ s'exécutent concurremment. Si la valeur cumulée des demandes maximales de temps de processeur de tous ces modules, pour tout intervalle, ne dépasse pas la durée de cet intervalle, le système est alors faisable.

Lemme 3.3. *Soit un système exécutant un ensemble de modules $Modules$. Il est ordonnançable sous* EDF *sur un processeur si :*

$$1. \quad \sum_{M \in Modules} \max_{m \in Modes[M]} \{U(m)\} \leq 1, \ et \tag{3.16}$$

$$2. \quad \sum_{M \in Modules} mdbf_M(\Delta) \leq \Delta \qquad \forall\ \Delta > 0 \tag{3.17}$$

Démonstration. La première condition, dont la nécessité a été prouvée par Leung et Merrill dans [107], exprime que le taux d'utilisation ne peut pas être supérieur à 1. La suffisance de la seconde condition peut être démontrée de la même manière que la condition d'ordonnançabilité pour les tâches à requête avant échéance formulée par Baruah dans [24]. On procède par l'absurde. On suppose que la seconde condition est satisfaite et que l'ordonnancement n'est pas faisable. Soit donc τ une tâche dont l'échéance est dépassée à un instant t_2. Cette tâche peut être préemptée par des tâches dont les échéances sont inférieures ou égales à t_2. Le dépassement d'une échéance survient suite à une période ininterrompue d'activité du processeur. Soit t_1 le dernier instant antérieur à t_2 tel qu'aucune tâche avec échéance dans l'intervalle $[t_1, t_2]$ ne s'exécute. Alors, une tâche est activée à l'instant t_1. A chaque instant dans l'intervalle $[t_1, t_2]$ il doit y avoir une tâche qui s'exécute et dont la date d'activation n'est pas antérieure à t_2. Ces tâches peuvent appartenir à des modules différents. Dans l'intervalle $[t_1, t_2]$ la demande maximale de temps de processeur faite par les tâches de module M ne dépasse pas $mdbf_M(t_2 - t_1)$ (voir Définition 3.11). Puisqu'à l'instant t_2 la tâche τ dépasse son échéance suite à une période continue d'activité de processeur initiée à l'instant t_1, on doit avoir $\sum_{M \in Modules} mdbf_M(t_2 - t_1) > t_2 - t_1$. Cela contredit l'hypothèse. □

La condition ci-dessus est seulement suffisante. Cela vient du fait qu'elle ne garantit pas que les traces d'exécution qui produisent les plus grandes charges du processeur dans différents modules commencent au même instant. Rappelons que, pour les mêmes raisons, comme montré dans la section 2.7, les méthodes d'analyse pour les systèmes à déclenchement événementiel ne sont pas en mesure de donner une réponse exacte dans le contexte de systèmes dirigés par le temps.

3.5.2 Borne de faisabilité

L'intervalle temporel sur lequel appliquer les conditions du Lemme 3.3 doit être borné. En effet, à défaut d'une telle limite, il n'est pas possible de tester exhaustivement l'ordon-

nançabilité du système. En section 2.4.2 la méthode de calcul de borne pour une longueur maximale des intervalles à vérifier dans le cas de *Recurring Branching Task* (l'Equation 2.21) [25] a été présentée. C'est le même procédé qui est utilisé ici. Il se résume comme suit. Tout d'abord, la demande maximale de temps processeur est bornée en fonction de la longueur de l'intervalle dans lequel cette demande est faite. La borne sur la demande maximale de temps de processeur est ensuite comparée à la durée de l'intervalle. En résolvant cette inégalité, la durée maximale de l'intervalle dans lequel le dépassement d'une des échéances peut se produire si le système est infaisable est alors obtenue.

Chaque intervalle Δ dans lequel on observe l'exécution d'une trace d'un module peut être considéré comme la composition de trois intervalles adjacents : Δ_1, Δ_2 et Δ_3. Ces intervalles sont définis ainsi :

— Δ_2 s'étale entre le premier et le dernier instant d'hyperpériodes de mode dans l'intervalle Δ,

— Δ_1 commence à l'instant de début de Δ et se termine à l'instant de début de Δ_2,

— Δ_3 s'étend entre la fin de Δ_2 et la fin de Δ.

La Figure 3.5 illustre cette décomposition.

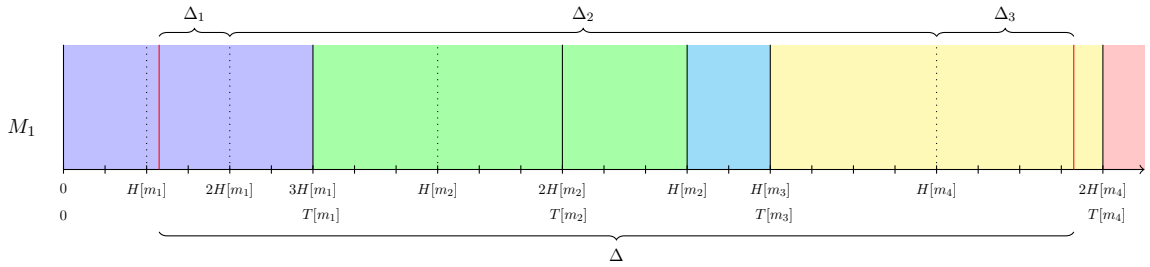


FIGURE 3.5 – Décomposition d'une trace sur un intervalle Δ en intervalles Δ_1, Δ_2 et Δ_3

$df_M(\Delta_1)$, $df_M(\Delta_2)$ et $df_M(\Delta_3)$ dénotent les valeurs des fonctions de demande des traces du module M qui s'exécutent respectivement dans les intervalles Δ_1, Δ_2 et Δ_3. D'après le Lemme 3.1, toute instance de tâche dont la date d'activation se trouve dans un de ces intervalles a sa date d'échéance dans le même intervalle. La tâche finit toujours son exécution dans l'intervalle dans lequel elle a commencé à s'exécuter. Si $df_M(\Delta)$ désigne la valeur de la fonction de demande d'une trace du module M dans un intervalle Δ, elle peut s'exprimer par :

$$df_M(\Delta) = df_M(\Delta_1) + df_M(\Delta_2) + df_M(\Delta_3) \tag{3.18}$$

La valeur de la fonction de demande dans ce premier intervalle Δ_1 peut être bornée supérieurement par :

$$df_M(\Delta_1) \leq \max_m df_M(m, r_{\min}(m), H[m]) \leq \max_m (U(m)H[m]) \tag{3.19}$$

où $r_{\min}(m)$ est la première date d'activation d'une tâche de mode $m \in Modes[M]$ après le début de ce mode ; si une telle date n'existe pas, $r_{\min}(m)$ est égale à 0 (par exemple, une seule tâche s'exécute dans un mode et sa période ainsi que son Temps d'Exécution Logique sont égaux à la période de ce mode).

De même, dans le dernier intervalle Δ_3 la valeur de la fonction de demande est bornée supérieurement comme suit :

$$df_M(\Delta_3) \ \leq \ \max_m df_M(m, 0, d_{\max}(m)) \ \leq \ \max_m(U(m)H[m]) \tag{3.20}$$

où $d_{\max}(m)$ est la dernière date d'échéance d'une tâche de mode $m \in Modes[M]$ avant la fin de l'hyperpériode de ce mode ; si une telle date n'existe pas, $d_{\max}(m)$ est égale à $H[m]$.

Une trace qui s'exécute dans l'intervalle Δ_2 couvre les exécutions entières de plusieurs modes. La demande faite par les tâches d'une telle trace ne dépasse pas la valeur suivante :

$$df_M(\Delta_2) \ \leq \ \Delta \max_m U(m) \tag{3.21}$$

Les trois relations obtenues ci-dessus permettent, pour tout module M et tout intervalle Δ, de borner la valeur de la fonction $mdbf_M(\Delta)$. Supposons qu'une trace de module M qui s'est exécutée dans l'intervalle Δ produit la plus grande demande parmi tous les autres intervalles de même durée. En appliquant les formules 3.19, 3.20 et 3.21 à l'expression 3.18 et en remplaçant df_M par $mdbf_M$ on obtient :

$$mdbf_M(\Delta) \ \leq \ \Delta \max_m U(m) + 2 \max_m(U(m)H[m]) \tag{3.22}$$

Si le système n'est pas ordonnançable, la seconde condition du Lemme 3.3 n'est pas vérifiée. Il existe Δ tel que :

$$\sum_{M \in Modules} mdbf_M(\Delta) \ > \ \Delta \tag{3.23}$$

Grâce à l'expression 3.22 on obtient :

$$\sum_{M \in Modules} \left(\Delta \max_{m \in Modes[M]} U(m) \ + \ 2 \max_{m \in Modes[M]} (U(m)H[m]) \right) \ > \ \Delta \tag{3.24}$$

ce qui donne, en regroupant les termes :

$$\Delta \ < \ \frac{2 \times \sum_{M \in Modules} \max_{m \in Modes[M]} (U(m)H[m])}{1 - \sum_{M \in Modules} \max_{m \in Modes[M]} U(m)} \tag{3.25}$$

Lors de l'analyse de faisabilité, il est donc suffisant de vérifier les intervalles dont la longueur est inférieure à la valeur obtenue ci-dessus.

Définition 3.12 (**Borne de faisabilité**). *Pour un ensemble de modules $Modules$, la borne de faisabilité $\Delta_{max}(Modules)$ désigne la plus grande valeur de Δ qui satisfait l'inégalité 3.25.*

La borne tend vers l'infini pour un ensemble de modules $Modules$ tel que :

$$\sum_{M \in Modules} \max_{m \in Modes[M]} U(m) = 1$$

3.5.3 Condition d'ordonnançabilité d'un système E-TDL pour EDF

Le théorème suivant peut être obtenu en combinant le Lemme 3.3 avec la formule 3.25 pour la longueur maximale d'intervalle à vérifier. Ce théorème peut être considéré comme une condition suffisante d'ordonnançabilité pour *E-TDL* sous *EDF*.

Théorème 2 (**Ordonnançabilité d'E-TDL : conditions suffisantes**). *Soit un système exécutant un ensemble $Modules$ de modules. Il est ordonnançable sous* EDF *sur un processeur si :*

$$1. \quad \sum_{M \in Modules} \max_{m \in Modes[M]} \{U(m)\} \leq 1, \ et \tag{3.26}$$

$$2. \quad \sum_{M \in Modules} mdbf_M(\Delta) \leq \Delta \qquad \forall \Delta \ : \ 0 < \Delta \leq \Delta_{max}(Modules) \tag{3.27}$$

Démonstration. En appliquant la formule 3.25 au Lemme 3.3. □

Le système de l'Exemple 2.11 permet d'exprimer, en termes propres à *E-TDL*, pourquoi le Théorème 2 ne constitue qu'une condition suffisante pour la faisabilité de ce système.

Exemple 3.5. *Soit le système de l'Exemple 2.11. La condition 3.27 est vérifiée pour une longueur d'intervalle $\Delta = 8$. Après analyse de toutes les traces de cette durée les valeurs de demande maximale de temps de processeur suivantes sont obtenues :*

- $mdbf_{M_1}(8) = 7$ *pour la trace* $\sigma_{M_1} = (m'_1, 3, 4, (m_1, 1), 3)$,
- $mdbf_{M_2}(8) = 2$ *pour la trace* $\sigma_{M_2} = (m_2, 0, 8, \emptyset, \emptyset, 0)$.

La demande cumulée de modules M_1 et M_2 dépasse la longueur de l'intervalle et la condition 3.27 n'est pas remplie :

$$\sum_{M \in Modules} mdbf_M(8) = mdbf_{M_1}(8) + mdbf_{M_2}(8) = 9 > 8$$

En fait, les traces σ_{M_1} et σ_{M_2} ne s'exécutent jamais en même temps. Leurs demandes ne s'additionnent donc pas et le système est faisable. Le Théorème 2 s'applique mal car il ne se limite pas aux seules traces qui s'exécutent à partir du même point dans le temps. Les fonctions de borne de demande maximale pour tous les intervalles de cet exemple sont tracées sur la Figure 3.6.

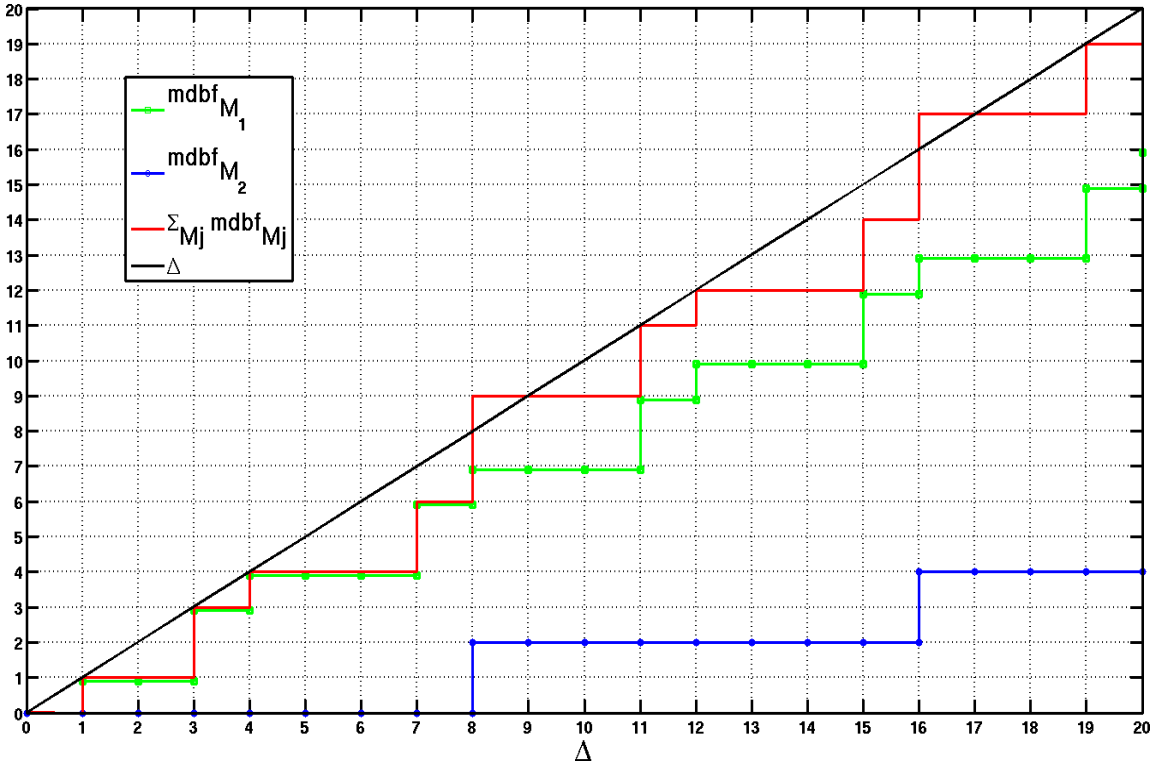


FIGURE 3.6 – Fonctions des bornes de demande maximale pour le système de l'Exemple 3.5.

3.6 Configurations parallèles

Afin de réduire le pessimisme du Théorème 2 et de formuler une condition nécessaire et suffisante d'ordonnançabilité pour *E-TDL*, il est nécessaire de définir, pour chaque trace d'exécution, un ensemble d'instants auxquels cette trace peut potentiellement commencer. En estimant la demande totale de tous les modules du système, les temps de processeur nécessaires pour les traitements des traces dans ces modules peuvent être additionnés s'il existe un instant auquel toutes ces traces commencent simultanément leurs exécutions. La suite aborde la question de savoir quels sont les instants de début d'exécution d'une trace et, par la suite, quelles sont les traces qui s'exécutent concurremment depuis un même instant.

Un observateur extérieur d'un système *E-TDL* pourrait décrire, à un instant donné, l'état d'exécution de celui-ci par un instantané indiquant les modes et les temps de ces modes exécutés à cet instant par les modules de ce système.

Définition 3.13 (**Configuration parallèle**). *Soit un système exécutant un ensemble Modules de modules E-TDL. A un instant donné, la configuration parallèle ζ de ce système*

est constituée par la collection ordonnée d'états de tous les modules $M_j \in Modules$ dans lesquels ces modules s'exécutent à cet instant.

$$\zeta = ((m_1, \delta_1), \ldots, (m_n, \delta_n)) \tag{3.28}$$

tel que : $\forall M_j \in Modules \;\; \exists (m_j, \delta_j) \in \zeta \, : \, m_j \in Modes[M_j]$ *; et* $n = |Modules|$.

Exemple 3.6. *Pour le système de l'Exemple 2.11 toutes les configurations parallèles qui sont atteignables appartiennent à l'ensemble suivant :*

$$\zeta \in \{((m_1, \delta_1), (m_2, \delta_2)), ((m'_1, \delta_1), (m_2, \delta_2)) \,|\, 0 \leq \delta_1 \leq 4,\, 0 \leq \delta_2 \leq 8,\, \delta_1 = \delta_2 \bmod 4\}$$

La configuration parallèle $\zeta = ((m'_1, 3), (m_2, 0))$*, prise en considération dans l'Exemple 3.5, ne se produit jamais.*

Afin de pouvoir retracer toutes les configurations dans lesquelles le système peut s'exécuter il faut d'abord identifier les instants de début de mode dans un module (section 3.6.1). Ensuite, il faut repérer la manière dont ces instants dans les modes de différents modules peuvent s'intercaler les uns par rapport aux autres (section 3.6.2). La connaissance de ces relations permet de reconstituer toutes les configurations parallèles atteignables qui dérivent des positionnement particuliers de démarrages des modes concurrents (section 3.6.3). Enfin, il se peut, sous certaines conditions, que toutes les configurations parallèles obtenues comme le produit des états de tous les modules soient atteignables. Dans cette situation, le résultat du Théorème 2 ne peut être amélioré (section 3.6.4).

3.6.1 Instant de début d'un mode

Soit (m, δ) un état du module M. Cet état est atteint après un temps δ à partir de l'instant de début de mode m, associé lui-même à l'état $(m, 0)$. Pour savoir à quels instants absolus $t \geq 0$ après le démarrage du système ($t = 0$) cet état peut être atteint, il est nécessaire de déterminer les instants absolus qui peuvent être ceux de l'entrée dans le mode m du module M. Farcas [59] propose pour cela d'étudier le graphe de changement de modes (voir section 3.2.1) ainsi que tous les chemins qui peuvent le parcourir.

L'ensemble des instants auxquels le module M peut entrer dans le mode m est désigné par $start(m)$. Cet ensemble est constitué par toutes les longueurs des chemins w depuis le mode initial jusqu'au mode m.

$$start(m) = \{\, |w| \mid head(w) = Init[M],\, tail(w) = m \,\} \tag{3.29}$$

Exemple 3.7. *Soit M un module tel que $Modes[M] = \{m_1, m_2, m_3, m_4\}$ et $m_1 = Init[M]$. Depuis le mode m_1 le module peut commuter vers les modes m_2 ou m_3, de m_2 vers m_4 et de m_3 vers m_4. Le module M peut atteindre le mode m_4 en empruntant tout chemin w parmi les ensembles de chemins suivants :*

— $\{\,((m_1,\mu_1),(m_2,\mu_2),(m_4,\mu_4)) \mid \mu_1,\mu_2 \in \mathbb{N}_+, \mu_4 \in \mathbb{N}_0\,\}$
— $\{\,((m_1,\mu_1),(m_3,\mu_3),(m_4,\mu_4)) \mid \mu_1,\mu_3 \in \mathbb{N}_+, \mu_4 \in \mathbb{N}_0\,\}$

La longueur de tout chemin w s'exprime par une des deux séries arithmétiques ci-dessous (voir Equation 3.6) :

— $\mu_1 T_{sw}(m_1,m_2) + \mu_2 T_{sw}(m_2,m_4) + \mu_4 T[m_4]$
— $\mu_1 T_{sw}(m_1,m_3) + \mu_3 T_{sw}(m_3,m_4) + \mu_4 T[m_4]$

Définition 3.14 (**Chaîne de base dans le graphe de changement de mode**). *Pour un module, une chaîne de base p, dans son graphe de changement de mode, est un ensemble de périodes de changements de modes et de périodes de modes qui peuvent être empruntées par un chemin w dans ce graphe tel que la période du mode $tail(w)$ est exécutée au moins une fois. Si $w = ((m_1,\mu_1),\ldots,(m_k,\mu_k),(m_{k+1},\mu_{k+1}),\ldots,(m_n,\mu_n))$ tel que $\mu_n \geq 1$ est un chemin dans un graphe de changement de mode, la chaîne de base p qui lui correspond est donnée par :*

$$p = \{\, T_{sw}(m_k,m_{k+1}) \mid 1 \leq k < n \,\} \cup \{\, T[m_k] \mid k = n \,\}. \tag{3.30}$$

Exemple 3.8. *Pour le module de l'Exemple 3.7, les chaînes qui mènent du mode m_1 à m_4 sont données ci-dessous :*

— $p_{m_1,m_2,m_4} = \{\, T_{sw}(m_1,m_2), T_{sw}(m_2,m_4), T[m_4] \,\}$
— $p_{m_1,m_3,m_4} = \{\, T_{sw}(m_1,m_3), T_{sw}(m_3,m_4), T[m_4] \,\}$

La définition suivante permet de trouver le plus grand facteur commun de toutes les longueurs de chemins construits à partir de la même chaîne de base.

Définition 3.15 (**Plus grand commun diviseur d'une chaîne de base**). *Le plus grand commun diviseur $pgcd(p)$ d'une chaîne de base $p = \{T_{sw}(m_1,m_2),\ldots,T[m_n]\}$ est le plus grand commun diviseur de toutes les périodes de changement de mode et de toutes les périodes de mode qui appartiennent à cette chaîne.*

$$pgcd(p) = pgcd(T_{sw}(m_1,m_2),\ldots,T[m_n]) \tag{3.31}$$

Exemple 3.9. *Soient les modes m_1,m_2,m_3,m_4 tels que introduits dans l'Exemple 3.7. On définit par ailleurs, les périodes de changements de modes et la période de mode m_4 comme suit : $T_{sw}(m_1,m_2) = 20$, $T_{sw}(m_2,m_4) = 25$, $T[m_4] = 15$. Le plus grand commun diviseur de la chaîne de base p_{m_1,m_2,m_4} est égal à :*

$$pgcd(p_{m_1,m_2,m_4}) = 5$$

On remarque que les longueurs de chemins basés sur cette chaîne constituent la séquence suivante :

$$45, 60, 65, 70, 75, 80, 85, 90, \ldots$$

A partir de 60 chaque entier multiple de 5 et donc multiple de $pgcd(p_{m_1,m_2,m_4})$ est une longueur de chemin qui traverse la chaîne p_{m_1,m_2,m_4} et, par conséquent, un possible instant de début du mode m_4.

Si p est une chaîne de base pour le chemin w, en introduisant $T'_{sw}(m_k, m_{k+1}) = \frac{T_{sw}(m_k,m_{k+1})}{pgcd(p)}$, la Formule 3.6 pour la longueur de ce chemin se récrit de la manière suivante :

$$|w| = \sum_{k=1}^{n} \mu_k \cdot T_{sw}(m_k, m_{k+1}) = pgcd(p) \cdot \sum_{k=1}^{n} \mu_k \cdot T'_{sw}(m_k, m_{k+1}). \tag{3.32}$$

Tous les nombres $T'_{sw}(m_k, m_{k+1})$ de l'équation ci-dessus sont premiers entre eux. Frobenius [142, 153] a démontré qu'étant donnés des entiers positifs $a_1, a_2, \ldots, a_n$ premiers entre eux, il existe un entier $F(a_1, \ldots, a_n)$ tel que les entiers supérieurs à celui-ci sont tous de la forme $k_1a_1 + k_2a_2 + \cdots + k_na_n$ pour des entiers positifs ou nuls $k_1, k_2, \ldots, k_n$. En d'autres termes, tout entier plus grand que $F(a_1, \ldots, a_n)$ peut être exprimé comme une combinaison linéaire à coefficients positifs ou nuls des entiers positifs $a_1, a_2, \ldots, a_n$. Pour une chaîne de base p, tout multiple assez grand de $pgcd(p)$ est une combinaison linéaire des périodes de changement de mode $T_{sw}(m_1, m_2), \ldots$ et de $T[m_n]$ incluses dans cette chaîne. Ce multiple est une longueur de chemin valide. Si p mène du mode initial au mode m, les multiples de $pgcd(p)$, à partir d'une certaine valeur, sont tous des instants possibles de début de mode m :

$$\exists n_0 \in \mathbb{N} : \forall n \in \mathbb{N}, n \geq n_0 \;\; n\, pgcd(p) \in start(m) \tag{3.33}$$

Il peut exister plusieurs chemins qui mènent au mode m en traversant des modes différents. Si P désigne l'ensemble de tous ces chemins, alors :

$$start(m) \subseteq \bigcup_{p \in P} \{\, n\, pgcd(p) \mid n \in \mathbb{N} \,\} \tag{3.34}$$

Alors, l'état (m, δ) du mode M peut se produire aux instants $t = t_{start}(m) + \delta$ tels que $t_{start}(m) \in start(m)$.

3.6.2 Intervalles de temps entre débuts de mode dans des modules distincts

Entre le problème posé ici et celui abordé par Pellizzoni et Lipari pour les tâches asynchrones, discuté dans la section 2.4.2, plusieurs similitudes existent. Les contextes étant différents, les résultats des travaux de Pellizzoni et Lipari ne sont pas directement applicables. Néanmoins, certaines de leurs observations permettent de reconnaître des mécanismes similaires et de fonder alors la proposition de solution au problème posé suivant.

Soient deux modules différents M_a et M_b ($M_a, M_b \in Modules$ et $M_a \neq M_b$). Le module M_a s'exécute en mode m_a ($m_a \in Modes[M_a]$) tandis que le module M_b s'exécute au même moment en mode m_b ($m_b \in Modes[M_b]$). Il s'agit de trouver tous les intervalles qui peuvent séparer les activations de ces deux modes.

Exemple 3.10. *Pour le système de l'Exemple 2.11, l'intervalle de temps entre le début de la plus récente activation de mode $m_1 \in Modes[M_1]$ et le début de la plus récente activation*

de mode $m_2 \in Modes[M_2]$ est égal soit à 0 soit à 4. Les configurations parallèles suivantes peuvent être obtenues :

$$\{\, ((m_1, \delta_1), (m_2, \delta_2)) \,|\, \delta_1 = \delta_2 - 0 < 4 \,\}$$

si les deux modes ont commencé leurs activations les plus récentes en même temps, et

$$\{\, ((m_1, \delta_1), (m_2, \delta_2)) \,|\, \delta_1 = \delta_2 - 4 < 4 \,\}$$

si le mode m_2 a commencé 4 unités de temps avant la plus récente activation du mode m_1.

Lemme 3.4 (**Intervalle entre débuts de deux modes**). *L'intervalle de temps entre les instants de début du mode $m_a \in Modes[M_a]$ et du mode $m_b \in Modes[M_b]$, s'exécutant respectivement dans le module $M_a \in Modules$ et dans le module $M_b \in Modules$ tels que $M_a \neq M_b$, appartient à l'ensemble suivant :*

$$\bigcup_{p_a \in P_a, p_b \in P_b} \{\, \alpha \cdot pgcd\,(p_a \cup p_b) \,|\, \alpha \in \mathbb{Z} \,\}. \tag{3.35}$$

où P_a est l'ensemble des chemins du mode initial $Init[M_a]$ au mode m_a dans le module M_a, et P_b est l'ensemble des chemins du mode initial $Init[M_b]$ au mode m_b dans le module M_b et $p_a \cup p_b$ réunit les ensembles de périodes de changement de mode et de périodes de mode d'un chemin $p_a \in P_a$ et d'un chemin $p_b \in P_b$.

Démonstration. La différence $\Delta_{start}(p_a, p_b)$ entre les instants de début du mode m_a et du mode m_b atteints respectivement par le chemin p_a et par le chemin p_b s'exprime à partir de l'Equation (3.33) :

$$\Delta_{start}(p_a, p_b) = n_a \cdot pgcd(p_a) - n_b \cdot pgcd(p_b) \tag{3.36}$$

D'après l'identité de Bézout [36], pour tous les entiers non nuls c et d, il existe des entiers $x, y \in \mathbb{Z}$ tels que $c \cdot x + d \cdot y = pgcd(c, d)$ et chaque entier de la forme $c \cdot x + d \cdot y$ est un multiple de $pgcd(c, d)$. Cela implique que toutes les valeurs de $\Delta_{start}(p_a, p_b)$ sont des multiples de $pgcd\,(pgcd(p_a),\, pgcd(p_b))$. Puisque l'opération de plus grand commun diviseur est associative [69], le terme $pgcd\,(pgcd(p_a),\, pgcd(p_b))$ peut être récrit comme $pgcd\,(p_a \cup p_b)$. Pour déterminer tout intervalle possible entre les débuts des modes m_a et m_b, tous les chaînes conduisant à ces modes doivent être considérés.

□

Le lemme ci-dessus prend en compte tous les instants possibles de début des deux modes. L'analyse doit se borner uniquement aux instants de démarrage de modes qui s'exécutent en même temps.

Lemme 3.5 (**Intervalle entre débuts de deux modes concurrents**). *Soient le mode* $m_a \in Modes[M_a]$ *et le mode* $m_b \in Modes[M_b]$ *dans les modules* $M_a \in Modules$ *et* $M_b \in Modules$ *tels que* $M_a \neq M_b$. *Si l'instance courante de mode* m_b *n'a pas commencé après l'instance courante de mode* m_a, *l'intervalle entre les instants de début de ces deux instances prend ses valeurs dans :*

$$\bigcup_{p_a \in P_a, p_b \in P_b} \{\, \alpha \cdot pgcd(p_a \cup p_b) \,|\, \alpha \in \mathbb{N} : 0 \leq \alpha \cdot pgcd(p_a \cup p_b) < T[m_b] \,\} \tag{3.37}$$

Démonstration. Si les plus récentes instances de mode m_a et m_b commencent en même temps alors $\alpha = 0$ donne la longueur nulle de l'intervalle qui sépare leurs démarrages. Selon le Lemme 3.4, les démarrages potentiels du mode m_b peuvent avoir lieux à tous les instants décalés d'un multiple de $pgcd(p_a \cup p_b)$ par rapport à l'instant de démarrage du mode m_a. Néanmoins, au delà d'une certaine valeur de α, les dates de démarrage de mode m_b coïncident avec celles de ses instances antérieures (qui ne sont plus actives). L'instance de mode m_b qui est concurrente avec l'instance courante de mode m_a ne commence pas plus tôt que $T[m_b]$ avant la date d'activation de l'instance courante de m_a. Alors, la valeur de $\alpha\, pgcd(p_a \cup p_b)$ ne peut pas dépasser $T[m_b]$. □

Le lemme précédent s'étend à k modes concurrents, $k \geq 3$. A cet effet, il faut tenir compte des relations entre tous les modes dans une configuration donnée.

Exemple 3.11. *Soient trois modes* $m_1 \in Modes[M_1]$, $m_2 \in Modes[M_2]$, $m_3 \in Modes[M_3]$ *s'exécutant dans trois modules différents* $M_1, M_2, M_3 \in Modules$ *tels que* $M_1 \neq M_2 \neq M_3$. *Les modes* m_1, m_2, m_3 *sont respectivement atteignables par les chemins* p_1, p_2 *et* p_3. *Par*

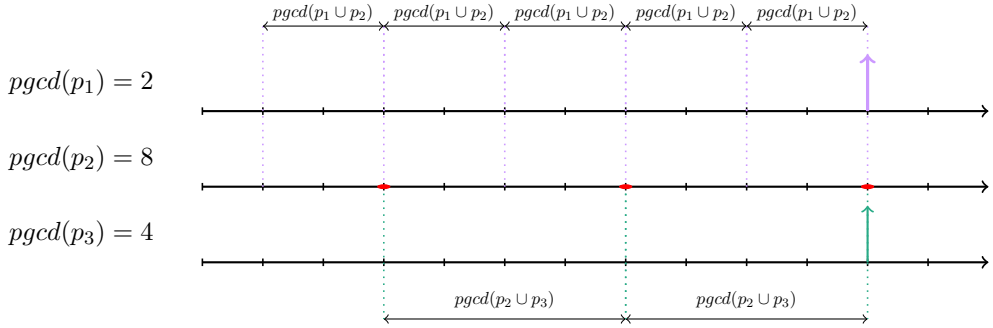


FIGURE 3.7 – Points de démarrage du mode m_2 de l'Exemple 3.11.

ailleurs, les plus grands communs diviseurs sont égaux à $pgcd(p_1) = 2$, $pgcd(p_2) = 8$ *et* $pgcd(p_3) = 4$.

Le Lemme 3.4 indique que les modes m_2 *et* m_3 *peuvent commencer* $2n$ *unités de temps après ou avant le mode* m_1 *(car* $pgcd(p_1 \cup p_2) = 2$ *et* $pgcd(p_1 \cup p_3) = 2$*). Néanmoins, si les*

modes m_1 et m_3 commencent en même temps, il n'est plus vrai que, dans cette configuration particulière, le mode m_2 puisse commencer chaque $2n$ unités de temps après ou avant le début du mode m_1. Les intervalles de temps entre les débuts du mode m_2 et du mode m_3 doivent être des multiples de 4. En effet, pour déterminer les instants de début du mode m_2 les deux relations doivent être prises en compte : celle entre les modes m_1 et m_2 ainsi que celle entre les modes m_2 et m_3.

Lemme 3.6 (**Intervalle entre débuts de k modes concurrents**). *Soit $Modules' = \{M_1, \ldots, M_{k-1}\} \subsetneq Modules$ un ensemble de $k-1$ modules où $3 \leq k \leq |Modules|$. Soit aussi, $\forall j < k$, un mode $m_j \in Modes[M_j]$.*

Soit encore $M_k \in Modules - Modules'$ et $m_k \in Modes[M_k]$.

Supposons que toutes les instances des modes $m_1, \ldots, m_{k-1}$ et m_k s'exécutent en même temps et que chaque mode m_j tel que $1 \leq j \leq k$ commence son exécution en empruntant le chemin p_j. Supposons encore que la plus récente activation de mode m_1 n'ait lieu avant aucune autre plus récente activation de mode m_j tel que $1 < j \leq k$ ($\forall j > 1 : \delta_1 \leq \delta_j$).

Si chaque mode $m_j \in Modules'$ ($1 < j < k$) a commencé sa plus récente instance d'exécution $\alpha_j \, pgcd(p_1 \cup p_j)$ unités de temps avant la plus récente instance d'exécution de mode m_1, tout intervalle entre les plus récentes activations de mode m_1 et m_k est donné par $\alpha_k \, pgcd(p_1 \cup p_k)$ pour tout $\alpha_k \in \mathbb{N}$ et respecte les conditions suivantes :

1. $0 \leq \alpha_k \cdot pgcd(p_1 \cup p_k) < T[m_k]$ (3.38)
2. $\forall j \in \mathbb{N} : 1 < j < k, \exists r \in \mathbb{Z} :$
$$\alpha_k \cdot pgcd(p_1 \cup p_k) = \alpha_j \cdot pgcd(p_1 \cup p_j) + r \cdot pgcd(p_j \cup p_k) \quad (3.39)$$

Démonstration. La première condition résulte du Lemme 3.5. La seconde condition ancre l'instant de début de mode m_k par rapport aux instants de début des autres modes. Tout mode m_j tel que $1 < j < k$ commence son instance courante $\alpha_j \cdot pgcd(p_1 \cup p_j)$ unités de temps avant l'instance courante du mode m_1. Le mode m_k commence son instance courante $\alpha_k \cdot pgcd(p_1 \cup p_k)$ unités de temps avant l'instance courante du mode m_1. La distance entre les débuts des modes m_1 et m_k est égale à $\alpha_k \cdot pgcd(p_1 \cup p_k) - \alpha_j \cdot pgcd(p_1 \cup p_j)$. D'après le Lemme 3.4, les instants de début de mode m_j et m_k sont séparés par un intervalle de temps $r \cdot pgcd(p_j \cup p_k)$ tel que $r \in \mathbb{Z}$. □

Exemple 3.12. *Soient les mêmes modules, les mêmes modes et les mêmes chemins que dans l'Exemple 3.11, en précisant que $T[m_3] = 12$. Par le Lemme 3.6, on sait que les débuts de mode m_1 et m_2 peuvent être séparés par des intervalles de temps qui sont des multiples de 2. En supposant que la dernière instance du mode m_2 commence 2 unités de temps avant le début de la dernière instance de mode m_1 : $pgcd(p_1 \cup p_2) = 2$, $\alpha_2 \cdot pgcd(p_1 \cup p_2) = 2$ pour $\alpha_2 = 1$. Etant donnée cette configuration ($\alpha_2 = 1$) de démarrage des modes m_1 et m_2, on cherche maintenant tous les intervalles possibles entre les instants des débuts des modes m_1 et m_3 si ce dernier ne commence pas après le mode m_1.*

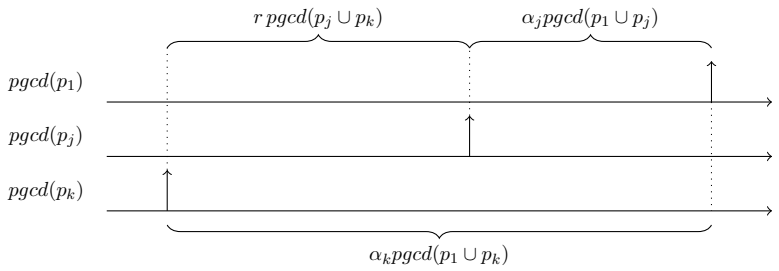


FIGURE 3.8 – Illustration de la preuve du Lemme 3.6.

La première condition du Lemme 3.6 prend la forme suivante :

$$0 \leq \alpha_3 \cdot pgcd(p_1 \cup p_3) < T[m_3] \;\; \Rightarrow \;\; 0 \leq \alpha_3 \cdot 2 < 12 \;\; \Rightarrow \;\; 0 \leq \alpha_3 < 6$$

Pour tous les α_3 qui satisfont la première condition, on évalue la seconde condition :

$$\alpha_3 \cdot pgcd(p_1 \cup p_3) = \alpha_2 \cdot pgcd(p_1 \cup p_2) + r \cdot pgcd(p_2 \cup p_3) \;\; \Rightarrow \;\; \alpha_3\, 2 = 2 + r\, 4$$

ce qui donne les équations suivantes :

$$\begin{aligned}
0 = 2 + 4r &\;\; \Rightarrow \;\; r = -0.5 && (\alpha_3 = 0)\\
2 = 2 + 4r &\;\; \Rightarrow \;\; r = 0.0 && (\alpha_3 = 1)\\
4 = 2 + 4r &\;\; \Rightarrow \;\; r = 0.5 && (\alpha_3 = 2)\\
6 = 2 + 4r &\;\; \Rightarrow \;\; r = 1.0 && (\alpha_3 = 3)\\
8 = 2 + 4r &\;\; \Rightarrow \;\; r = 1.5 && (\alpha_3 = 4)\\
10 = 2 + 4r &\;\; \Rightarrow \;\; r = 2.0 && (\alpha_3 = 5)
\end{aligned}$$

Le coefficient r prend des valeurs entières pour $\alpha_3 = 1$, $\alpha_3 = 3$ et $\alpha_3 = 5$. Alors, l'intervalle entre les instants de démarrages de modes m_1 et m_3, donné par $\alpha_3\, pgcd(p_1 \cup p_3)$, peut être égal à 2, 6 ou 10.

Dans l'exemple précédent il apparaît que les valeurs de α_k pour lesquelles les conditions du Lemme 3.6 sont vérifiées observent une certaine régularité. Le Lemme 3.7 détermine l'intervalle entre toutes les valeurs de α_k qui satisfont les conditions de ce lemme. Pour une valeur de j donnée, la relation 3.39 est vérifiée par $\alpha_k \cdot pgcd(p_j, p_k)$ et $\alpha'_k \cdot pgcd(p_j, p_k)$ tels que $\alpha'_k > \alpha_k$ si la différence $(\alpha'_k \cdot pgcd(p_j, p_k) - \alpha_k \cdot pgcd(p_j, p_k))$ est un multiple de $pgcd(p_j \cup p_k)$. Cette observation s'étend sur tous les j tels que $1 < j < k$.

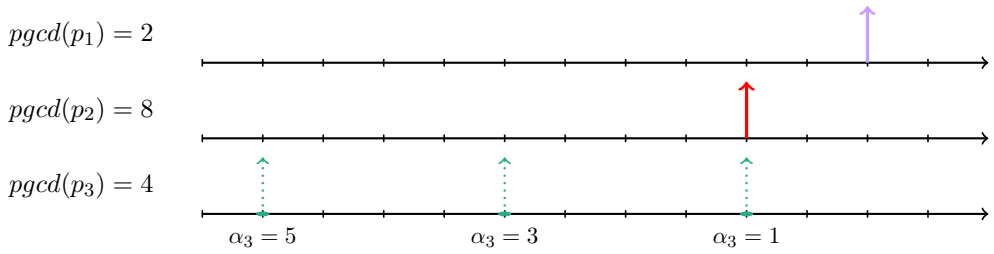


FIGURE 3.9 – Points de démarrage du mode m_3 de l'Exemple 3.12.

Lemme 3.7. *Si les conditions du Lemme 3.6 sont satisfaites pour α_k, sa seconde condition est aussi satisfaite pour $\alpha'_k = \alpha_k + \Delta_k$ tel que :*

$$\Delta_k = \frac{ppcm\left(\bigcup_{j=1}^{k-1} pgcd\left(p_j \cup p_k\right)\right)}{pgcd(p_1 \cup p_k)} \tag{3.40}$$

Démonstration. L'équation 3.39 est vérifiée par α_k pour tout j tel que $1 < j < k$. Cela implique, pour un tel k, que :

$$(\alpha_k \cdot pgcd(p_1 \cup p_k) - \alpha_j \cdot pgcd(p_1 \cup p_j)) \bmod pgcd(p_j \cup p_k) = 0 \tag{3.41}$$

La valeur suivante $\alpha_k + \Delta_{j,k}$, pour $\Delta_{j,k} > 0$, qui vérifie l'équation 3.39 pour le même j doit, par conséquent, satisfaire la condition ci-dessous :

$$((\alpha_k + \Delta_{j,k}) \cdot pgcd(p_1 \cup p_k) - \alpha_j \cdot pgcd(p_1 \cup p_j)) \bmod pgcd(p_j \cup p_k) = 0 \tag{3.42}$$

Par la formule 3.41, $(\alpha_k \cdot pgcd(p_1 \cup p_k) - \alpha_j \cdot pgcd(p_1 \cup p_j))$ est divisible par $pgcd(p_j \cup p_k)$, alors $\Delta_{j,k} \cdot pgcd(p_1 \cup p_k)$ doit être également divisible par $pgcd(p_j \cup p_k)$:

$$\Delta_{j,k} \cdot pgcd(p_1 \cup p_k) \bmod pgcd(p_j \cup p_k) = 0 \tag{3.43}$$

On cherche la plus petite valeur de $\Delta_{j,k}$ pour laquelle la condition ci-dessus est vraie. Alors, $\Delta_{j,k} \cdot pgcd(p_1 \cup p_k)$ doit être égal au plus petit commun multiple de $pgcd(p_1 \cup p_k)$ et $pgcd(p_j \cup p_k)$:

$$\Delta_{j,k} \cdot pgcd(p_1 \cup p_k) = ppcm(pgcd(p_1 \cup p_k), pgcd(p_j \cup p_k)) \tag{3.44}$$

On obtient :

$$\Delta_{j,k} = \frac{ppcm\left(pgcd\left(p_1 \cup p_k\right), pgcd\left(p_j \cup p_k\right)\right)}{pgcd(p_1 \cup p_k)} \tag{3.45}$$

α'_k est une valeur suivante qui vérifie la seconde condition du Lemme 3.6 pour tous les j si $\Delta_k = \bigcup_{1<j<k} \Delta_{j,k}$. Alors, Δ_k est le plus petit commun multiple des $\Delta_{j,k}$ pour j variant de 2 à $k-1$:

$$\Delta_k = ppcm\left(\bigcup_{j=2}^{k-1} \frac{ppcm\left(pgcd\left(p_1 \cup p_k\right), pgcd\left(p_j \cup p_k\right)\right)}{pgcd(p_1 \cup p_k)}\right) \tag{3.46}$$

On multiple l'équation ci-dessus par $pgcd(p_1 \cup p_k)$ et, grâce à la relation : $m\, ppcm(a,b) = ppcm(ma, mb)$, on obtient :

$$\Delta_k \cdot pgcd(p_1 \cup p_k) = ppcm\left(\bigcup_{j=2}^{k-1} ppcm\left(pgcd\left(p_1 \cup p_k\right), pgcd\left(p_j \cup p_k\right)\right)\right) \tag{3.47}$$

Puisque $ppcm(ppcm(a,b),c) = ppcm(a,b,c)$, cette équation se réduit à :

$$\Delta_k \cdot pgcd(p_1 \cup p_k) = ppcm\left(\bigcup_{j=1}^{k-1} pgcd\left(p_j \cup p_k\right)\right) \tag{3.48}$$

En divisant l'équation ci-dessus par $pgcd(p_1 \cup p_k)$ on obtient l'Expression 3.40 □

Les Lemmes 3.5 et 3.6 permettent de déterminer toutes les configurations de démarrage de modes s'exécutant concurremment dans des modules différents. Le procédé est décrit en détail dans le chapitre suivant, où une solution algorithmique est proposée (voir section 4.4). De façon générale, il s'agit tout d'abord de calculer, grâce au Lemme 3.5, les intervalles entre débuts de mode pour les modes des deux premiers modules de l'ensemble $Modules$, puis, en appliquant le Lemme 3.6, d'ajouter, un par un, les modes des modules suivants. La configuration de démarrage des modes est donnée par tous les coefficients $\alpha_2, \ldots, \alpha_n$ qui satisfont les conditions des Lemmes 3.5 et 3.6 pour un ensemble de n modules.

Définition 3.16 (**Ensemble des configurations possibles des démarrages de modes**). *Soit $Modules$ un ensemble de n modules et soient $m_1, \ldots, m_n$ n modes appartenant chacun à un module différent de cet ensemble ($\forall M_k \in Modules : \exists m_k \in Modes[M_k]$). L'exécution courante de chaque mode m_k est précédée depuis le mode initial de son module par le chemin p_k. L'ensemble des configurations possibles des démarrages de modes $\alpha^{(n)}(p_1, \ldots, p_n)$ pour les chemins $p_1, \ldots, p_n$ est l'ensemble de toutes les collections ordonnées de coefficients $\alpha_2, \ldots, \alpha_n$ pour lesquels les conditions des Lemmes 3.5 et 3.6 sont satisfaites. Chacune de ces collections décrit les intervalles entre les démarrages de modes qui s'exécutent en parallèle à un instant donné.*

Exemple 3.13. *Soient les mêmes modes, les mêmes modules et les mêmes chemins que dans les Exemples 3.11 et 3.12. Si la période du mode m_2 est égale à $T[m_2] = 8$ et la période du mode m_3 est égale à $T[m_3] = 12$, alors l'ensemble $\alpha^{(3)}(p_1, p_2, p_3)$ est donné par :*

$$\alpha^{(3)}(p_1, p_2, p_3) = \{(0,0), (0,2), (0,4), (1,1), (1,3), (1,5), (2,0), (2,2), (2,4), (3,1), (3,3), (3,5)\}$$

Il est possible que le même mode puisse être atteint par des chemins dont les chaînes comprennent des modes différents et, par conséquent, les plus grands communs diviseurs de ces chemins sont également différents. Dans ce cas, afin d'obtenir une image complète de tous les intervalles possibles entre démarrages de mode dans les différents modules, chaque séquence de chemins $(p_1, \ldots, p_n) \in \prod_{k=1}^{n} P_k$, où P_k est l'ensemble de tous les chemins qui mènent depuis le mode initial au mode m_k, doit être vérifiée.

3.6.3 Caractérisation d'une configuration parallèle

La connaissance des relations qui lient les dates de débuts des modes s'exécutant concurremment permet de reconstituer toutes les configurations parallèles atteignables par le système. A chaque configuration parallèle est associé un ensemble de traces qui peuvent démarrer simultanément dans tous les modules. Ces traces seront considérées pendant l'évaluation de la demande de temps de processeur cumulée par tous les modules du système.

Lemme 3.8 (**Observabilité de Configuration Parallèle**). *Soit un système composé de n modules. La configuration parallèle $\zeta = ((m_1, \delta_1), \ldots, (m_n, \delta_n))$ peut être observée pendant l'exécution de ce système s'il existe $(\alpha_2, \ldots, \alpha_n) \in \alpha^{(n)}(p_1, \ldots, p_n)$ tel que :*

$$\forall k, 1 < k \leq n,\ (\delta_k - \delta_1 + T[m_1]) \bmod T[m_k] = \alpha_k \cdot pgcd(p_1 \cup p_k) \tag{3.49}$$

Démonstration. Supposons que les modules ne changent pas de mode. Alors, après $T[m_1] - \delta_1$ unités de temps, le mode m_1 recommence son exécution. A cet instant, le redémarrage du mode m_1 n'a lieu avant aucun autre redémarrage des autres modes. Si les autres modules ne changent pas de mode, le temps de mode m_k à cet instant-là est égal à $\delta'_k = \delta_k - \delta_1 + T[m_1] \bmod T[m_k]$. L'intervalle de temps entre cette activation et l'activation la plus récente du mode m_k est égal à δ'_k. La longueur de cet intervalle doit satisfaire les conditions du Lemme 3.6. Donc $\forall\, k\ :\ 1 < k \leq n\ \ \exists\, \alpha_k : \delta'_k = \alpha_k \cdot pgcd(p_1 \cup p_k)$. □

3.6.4 Modes non-corrélés

Les paragraphes précédents cherchaient à caractériser les relations qui relient les états particuliers de différents modules. Dans ce qui suit, le problème inverse est abordé et l'on s'interroge sur les conditions pour qu'aucune relation n'existe entre ces états de telle sorte que ces états puissent éventuellement coïncider de manière arbitraire. Et dans ce cas, certaines traces d'exécution dans des modules différents peuvent être déclenchées au même instant. Leurs demandes de temps de processeur peuvent donc être additionnées quels que soient les états des modules à partir desquels ces traces débutent. Ainsi, dans ce cas, les conditions du Théorème 2 s'appliquent non seulement comme conditions suffisantes mais également nécessaires. Un parallèle peut être effectué avec l'analyse d'un ensemble de tâches

asynchrone en considérant cet ensemble comme synchrone. Si les périodes de ces tâches sont premières entre elles, le résultat de cette analyse est exact [148] (voir section 2.4.2).

L'exemple suivant illustre un cas où l'instant de démarrage d'un mode peut se produire à n'importe quel instant de tout autre mode concurrent.

Exemple 3.14. *Considérons trois modes m_1, m_2 et m_3 qui s'exécutent concurremment dans trois modules différents. Les plus grands communs diviseurs des chemins p_1, p_2, p_3 par lesquels les modules passent avant la plus récente activation de chacun de ces modes sont donnés respectivement par : $pgcd(p_1) = 2$, $pgcd(p_2) = 9$ et $pgcd(p_3) = 3$. Le mode m_1 peut donc commencer son exécution à tout instant des modes m_2 et m_3 car $pgcd(p_1) = 2$ est le premier avec $pgcd(p_2)$ et $pgcd(p_3)$. Par contre, la différence entre les instants de démarrage de ces deux derniers modes est toujours un multiple de 3.*

La condition ci-après permet d'établir, pour un ensemble de modes qui s'exécutent en parallèle, que tout temps d'un des modes de cet ensemble peut coïncider avec tous les temps des autres modes. Toute trace d'exécution initiée dans ce mode peut alors être déclenchée au même instant que toute autre trace dans un module différent. Ainsi, la demande de temps de processeur faite par chaque trace débutant dans ce mode peut devoir s'ajouter aux demandes des traces démarrant dans les modes concurrents.

Lemme 3.9 (**Mode non-corrélé**). *Soient n modes $m_1, \ldots, m_n$ s'exécutant en parallèle chacun dans un module différent. Chaque mode m_k pour tout k tel que $1 \leq k \leq n$ est atteignable par les chemins $p_k \in P_k$. Le temps δ_i du mode m_i peut coïncider avec tout temps de tout autre mode si et seulement si :*

$$\exists p_i \in P_i : \ \forall k \neq i,\ 1 \leq k \leq n, \exists p_k \in P_k,\ pgcd(p_i, p_k) = 1 \tag{3.50}$$

Démonstration. Soit δ_k le temps du mode m_k. L'intervalle entre les instants de début des modes m_i et m_k est égal à $(\delta_i - \delta_k) \bmod T[m_k]$. Selon le Lemme 3.5 cet intervalle est multiple de $pgcd(p_i, p_k)$. Pour tout $n \in \mathbb{N}$, $(\delta_i - \delta_k) \bmod T[m_k] = npgcd(p_i, p_k)$. Avec $pgcd(p_i, p_k) = 1$ on obtient $(\delta_i - \delta_k) \bmod T[m_k] = n$. Le temps de mode δ_k est égal à $(\delta_i + n) \bmod T[m_k]$. Cela implique que $\delta_k \in [0, T[m_k]]$. □

Il se peut que non seulement un mode, mais tous les modes concurrents, puissent s'exécuter dans toutes les configurations possibles de leurs temps de mode. Dans ce cas, toutes les traces de ces modes peuvent commencer leurs exécutions simultanément. Si la demande cumulative de ces traces ne satisfait pas les conditions du Théorème 2, il est inutile d'appliquer l'analyse plus détaillée.

Lemme 3.10 (**Cas synchrone**). *Soient n modes $m_1, \ldots, m_n$ s'exécutant en parallèle chacun dans un module différent. Pour tout k tel que $1 \leq k \leq n$, m_k est atteignable par tous les chemins $p_k \in P_k$. Toute configuration parallèle $\zeta = ((m_1, \delta_1), \ldots, (m_n, \delta_n))$ qui est dans*

l'ensemble de produit de tous les temps de ces modes ($\prod_{k=1}^{n}\{(m_k,\delta_k)\,|\,0 \leq \delta_k \leq T[m_k]\}$) *est observable si et seulement si :*

$$\exists P \in \prod_{k=1}^{n} P_k : \forall p_i, p_j \in P,\ pgcd(p_i, p_j) = 1 \tag{3.51}$$

3.7 Condition nécessaire et suffisante d'ordonnançabilité de plusieurs modules

Une configuration parallèle associe toutes les traces d'exécution qui peuvent débuter au même moment dans des modules différents. Le système est faisable s'il n'existe aucun intervalle de temps durant lequel la demande de temps de processeur dépasse le temps de processeur disponible. Dans chaque intervalle, on considère seulement les traces qui appartiennent à la même configuration parallèle et s'exécutent en même temps. Le théorème suivant donne les conditions nécessaires et suffisantes pour l'ordonnançabilité d'un système *E-TDL* sous *EDF* sur un processeur. Il réduit le pessimisme du Théorème 2.

Théorème 3 (**Ordonnançabilité d'un système E-TDL : conditions nécessaires et suffisantes**). *Soit un système exécutant un ensemble Modules de n modules E-TDL. Il est ordonnançable sous* EDF *sur un processeur si et seulement si :*

$$\sum_{k=1}^{n} \max_{m \in Modes[M_k]} \{U(m)\} \leq 1, \tag{3.52}$$

et pour chaque configuration parallèle $\zeta = ((m_1,\delta_1),\ldots,(m_n,\delta_n))$ *et chaque intervalle de temps* $0 < \Delta \leq \Delta_{max}(Modules)$ *:*

$$\sum_{k=1}^{n} \quad maxdf_{M_k}(m_k,\delta_k,\Delta) \leq \Delta \tag{3.53}$$

où (m_k,δ_k) *est un état du module* $M_k \in Modules$ *dans la configuration parallèle* ζ*.*

Démonstration. La preuve est adaptée des preuves du Lemme 3.3 et du Lemme 3.4 dans [24] pour l'ordonnançabilité des tâches à requête avant échéance. Avec les notations de la preuve du Lemme 3.3 et en procédant par l'absurde, on suppose que le système est infaisable dans l'intervalle $[t_1,t_2]$ et que la seconde condition du Théorème 3 est satisfaite. On suppose aussi qu'à l'instant t_1 le système s'exécute dans la configuration parallèle ζ qui appartient à l'ensemble de toutes les configurations parallèles observables. La demande de temps de processeur faite par les tâches de tout module $M_k \in Modules$ dans l'intervalle $[t_1,t_2]$ est donnée par la fonction $maxdf_{M_k}(m_k,\delta_k,\Delta)$ pour $\Delta = t_2 - t_1$ et $(m_k,\delta_k) \in \zeta$. Puisqu'à l'instant t_2 la tâche τ dépasse son échéance, la demande cumulative de temps de processeur faite par tous les modules de l'ensemble $Modules$ est plus grande que le temps de processeur disponible dans cet intervalle qui est égal à Δ. Par conséquent, $\sum_{k=1}^{n} maxdf_{M_k}(m_k,\delta_k,\Delta) > \Delta$, ce qui contredit l'hypothèse de départ. □

L'exemple suivant illustre les aspects les plus importants dans l'analyse d'ordonnançabilité basée sur les résultats obtenus dans ce chapitre.

Exemple 3.15. *Soit un système composé de trois modules* $Modules = \{M_1, M_2, M_3\}$ *E-TDL. Les modes et les commutateurs de mode au sein de chacun des modules sont indiqués sur la Figure 3.10 (le mode situé le plus haut dans chacun des modules est le mode initial). Le tableau 3.1 représente les périodes de mode et les périodes de changement de mode*

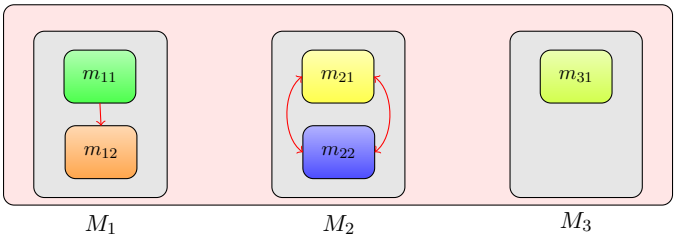


FIGURE 3.10 – Système composé de trois modules E-TDL

ainsi que les paramètres de toutes les tâches. Tout d'abord, le Théorème 2 est utilisé afin

Module	mode	T	T_{sw}	tâches
M_1	m_{11}	10	10	$\tau_{111}(0,2,6,10), \tau_{112}(0,1,5,5)$
	m_{12}	8	-	$\tau_{121}(2,1,2,8)$
M_2	m_{21}	4	4	$\tau_{211}(0,1,4,4)$
	m_{22}	8	8	$\tau_{221}(1,1,1,8)$
M_3	m_{31}	8	-	$\tau_{311}(2,1,1,8)$

TABLE 3.1 – Modes et paramètres de tâches

de vérifier l'ordonnançabilité du système. La somme des facteurs maximaux d'utilisation des modes dans tous les modules n'est pas plus grande que 1. La seconde condition est alors vérifiée. Sur la Figure 3.11 les fonctions des bornes de demande maximale $mdbf_{M_1}$, $mdbf_{M_2}$, $mdbf_{M_3}$ *sont tracées ainsi que leur somme comparée avec la durée d'intervalle de temps* Δ. *Comme le montre la Figure 3.11, la seconde condition du Théorème 2 n'est pas satisfaite pour* $\Delta = 1$ *et* $\Delta = 2$.

Pour $\Delta = 1$, *les demandes maximales de temps de processeur faites par les traces d'exécution de durée* 1 *au sein de modules* M_1, M_2 *et* M_3 *sont :*

- M_1 : *La demande de temps de processeur de toute trace d'exécution de durée* 1 *est égale à* 0 *:*

$$df(\sigma_{M_1}) = 0 \qquad si \ |\sigma_{M_1}| = 1.$$

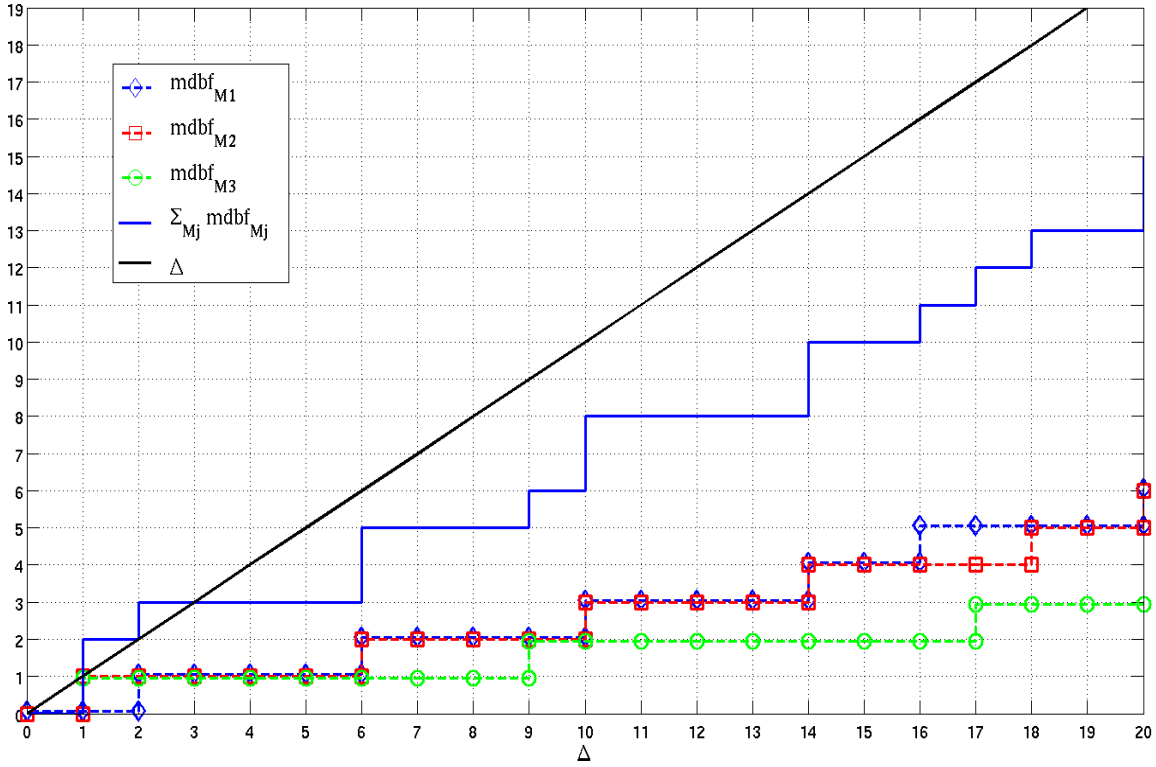


FIGURE 3.11 – Fonctions des bornes de demande maximale pour le système de l'Exemple 3.15.

- M_2 : *La demande de temps de processeur de trace* $(m_{22}, 1, 2, \emptyset, \emptyset, 0)$ *est égale à* 1. *Pour toute autre trace de durée* 1 *la demande de temps de processeur est égale à* 0 *:*

$$df(\sigma_{M_2}) = \begin{cases} 1 & si\ \ \sigma_{M_2} = (m_{22}, 1, 2, \emptyset, \emptyset, 0), \\ 0 & sinon \end{cases} \qquad si\ \ |\sigma_{M_2}| = 1.$$

- M_3 : *La demande de temps de processeur de trace* $(m_{31}, 2, 3, \emptyset, \emptyset, 0)$, *est égale à* 1. *Pour toute autre trace de durée* 1 *la demande de temps de processeur est égale à* 0 *:*

$$df(\sigma_{M_3}) = \begin{cases} 1 & si\ \ \sigma_{M_3} = (m_{31}, 2, 3, \emptyset, \emptyset, 0), \\ 0 & sinon \end{cases} \qquad si\ \ |\sigma_{M_3}| = 1.$$

Pour $\Delta = 2$, *les demandes maximales de temps de processeur faites par les traces d'exécution de cette durée sont présentées ci-dessous :*

- M_1 : *La demande de temps de processeur de trace* $(m_{12}, 2, 4, \emptyset, \emptyset, 0)$ *est égale à* 1.

Pour toute autre trace de durée 2 *la demande de temps de processeur est égale à* 0 *:*

$$df(\sigma_{M_1}) = \begin{cases} 1 & si\ \ \sigma_{M_1} = \sigma_{M_1} = (m_{12}, 2, 4, \emptyset, \emptyset, 0), \\ 0 & sinon \end{cases} \qquad si\ \ |\sigma_{M_1}| = 2.$$

- M_2 : *Les demandes de temps de processeur de trace* $(m_{22}, 1, 3, \emptyset, \emptyset, 0)$ *et* $(m_{22}, 0, 2, \emptyset, \emptyset, 0)$ *sont égales à* 1*. Pour toute autre trace de durée* 2 *la demande de temps de processeur est égale à* 0 *:*

$$df(\sigma_{M_2}) = \begin{cases} 1 & si\ \ \sigma_{M_2} \in \{(m_{22}, 1, 3, \emptyset, \emptyset, 0), (m_{22}, 0, 2, \emptyset, \emptyset, 0)\} \\ 0 & sinon \end{cases} \qquad si\ \ |\sigma_{M_2}| = 2.$$

- M_3 : *La demande de temps de processeur de trace* $(m_{31}, 2, 4, \emptyset, \emptyset, 0)$ *et* $(m_{31}, 1, 3, \emptyset, \emptyset, 0)$ *sont égales à* 1*. Pour toute autre trace de durée* 2 *la demande de temps de processeur est égale à* 0 *:*

$$df(\sigma_{M_3}) = \begin{cases} 1 & si\ \ \sigma_{M_3} \in \{(m_{31}, 2, 4, \emptyset, \emptyset, 0), (m_{31}, 1, 3, \emptyset, \emptyset, 0)\} \\ 0 & sinon \end{cases} \qquad si\ \ |\sigma_{M_3}| = 2.$$

A l'aide de l'analyse présentée dans la section 3.6, on peut vérifier s'il peut se produire que les traces indiquées ci-dessus démarrent en même temps. Pour cela on vérifie que les configurations parallèles qui correspondent aux démarrages de ces traces sont observables au sens du Lemme 3.8.

Pour $\Delta = 1$*, en tenant compte des traces identifiées ci-dessus et pour lesquelles le Théorème 2 ne garantit pas l'ordonnançabilité, chaque configuration parallèle dans laquelle ces traces débuteraient simultanément appartient à l'ensemble :*

$$\{((m_1, \delta_1), (m_{22}, 1), (m_{31}, 2)) \,|\, m_1 \in Modes[M_1],\, 0 \leq \delta_1 < T[m_1]\}$$

où (m_1, δ_1) *est un état quelconque du module* M_1*. Ces configurations décrivent un état d'exécution du système dans lequel les traces de longueur* 1 *effectuant la plus grande demande de processeur démarreraient, au même instant de temps absolu, simultanément. Ces configurations sont telles que, pour ce faire, le mode* m_{31} *doit avoir débuté une unité de temps avant le mode* m_{22}*.*

Dans un premier temps, au sein de chaque module, les chemins qui mènent du mode initial vers les modes de la configuration parallèle examinée doivent être identifiés. Ceux-ci se trouvent, accompagnés de leurs plus grands communs diviseurs, dans la Table 3.2 (le chemin vers m_{12} *n'est pas considéré pour* $\Delta = 1$*). Les plus grands communs diviseurs* $pgcd(p_a \cup p_b)$ *de paires de chemins qui mènent, dans des modules différents, vers des modes pouvant s'exécuter en parallèle sont donnés dans la Table 3.3. Ces valeurs seront utilisées pour vérifier les conditions des Lemmes 3.4 et 3.5. Selon le Lemme 3.4, pour* p_{22} *et* p_{31}*,*

	chemin	pgcd
p_{12}	$m_{11} \rightarrow m_{12}$	2
p_{22}	$m_{21} \rightarrow m_{22}$	4
p_{31}	m_{31}	8

TABLE 3.2 – Plus grands communs diviseurs des chemins

	p_{12}	p_{22}	p_{31}
p_{12}	-	2	2
p_{22}	2	-	4
p_{31}	2	4	-

TABLE 3.3 – Plus grands communs diviseurs pour les paires des chemins qui mènent aux modes concurrents

tout intervalle de temps séparant les débuts de ces deux modes est un multiple de $pgcd(p_{22} \cup p_{31}) = 4$. *Le second lemme est satisfait par les éléments de l'ensemble* $\alpha(p_{22}, p_{31}) = \{0, 1\}$. *Ce résultat s'interprète soit comme le début simultané des exécutions concurrentes des modes* m_{31} *et* m_{22} $(0 * pgcd(p_{22} \cup p_{31}))$, *soit comme le début de l'exécution de mode* m_{31} 4 *unités de temps avant le début de l'exécution concurrente de mode* m_{22} $(1 * pgcd(p_{22} \cup p_{31}))$. *Aucun coefficient de l'ensemble* $\alpha(p_{22}, p_{31})$ *ne satisfait la condition d'observabilité 3.49 pour l'état* $(m_{22}, 1)$ *du module* M_2 *et l'état* $(m_{31}, 2)$ *du module* M_3. *En effet,* $(2 - 1 + 8) \bmod 8 \notin \{0, 4\}$. *Par conséquent, la configuration parallèle dans laquelle ces deux états coïncident n'est pas observable. On en déduit que les traces d'exécution* σ_{M_2} *et* σ_{M_3} *telles que* $start(\sigma_{M_2}) = (m_{22}, 1)$ *et* $start(\sigma_{M_3}) = (m_{31}, 2)$ *ne commencent jamais au même moment. Donc, le système est faisable pour tous les intervalles de durée* $\Delta = 1$.

Pour $\Delta = 2$, *l'ensemble des configurations parallèles pour lesquelles la faisabilité du système n'est pas garantie par le Théorème 2 est défini par :*

$$\{((m_{12}, 2), (m_{22}, \delta_{22}), (m_{31}, \delta_{31})) \mid \delta_{22} = \{0, 1\},\ \delta_{31} = \{1, 2\}\}$$

Comme toutes ces configurations se rapportent à l'exécution concurrente des modes m_{12}, m_{22} *et* m_{31}, *les relations entre leurs instants de début sont à explorer. Tout d'abord, le Lemme 3.5 situe les chemins qui mènent aux modes* m_{12} *et* m_{22} *(voir Table 3.2) en supposant que l'instance courante du mode* m_{12} *n'est pas démarrée avant l'instance courante du mode* m_{22}. *On obtient les valeurs* $\alpha_2 \in \alpha^{(2)}(p_{12}, p_{22})$ $(\{0, 1, 2, 3\})$, *qui correspondent aux intervalles de longueurs* $\alpha_2 * pgcd(p_{12} \cup p_{22})$ $(0, 2, 4, 6)$ *entre les débuts des instances concurrentes des modes* m_{12} *et* m_{22}. *On intègre ensuite dans l'analyse le chemin qui mène au mode* m_{31}. *Le Lemme 3.6 permet d'obtenir l'ensemble* $\alpha^{(3)}(p_{12}, p_{22}, p_{31})$ *de paires* (α_2, α_3). *Chaque paire* (α_2, α_3) *décrit un schéma d'activation des modes dans lequel les modes* m_{22} *et* m_{31} *commencent leurs instances courantes respectivement* $\alpha_2 * \gcd(p_{12} \cup p_{22})$ *et* $\alpha_3 * \gcd(p_{12} \cup p_{31})$ *unités de temps avant le début de l'instance courante du mode* m_{12}. *Toutes ces paires* (α_2, α_3), *ainsi que des exemples de chemins correspondant à chacune de ces paires sont présentés dans la Table 3.4 (*w_{12} *est un chemin conduisant depuis le mode initial du Module* M_1 *vers le mode* m_{12}, w_{22} *est un chemin conduisant depuis le mode initial du Module* M_2 *vers le mode* m_{22} *et* w_{31} *est un chemin conduisant depuis le mode initial du Module* M_3 *vers le mode* m_{31} *; les chemins situés dans une même ligne de la table sont associés à une même paire* (α_2, α_3)*).*

(α_2, α_3)	exemple des chemins		
	w_{12}	w_{22}	w_{31}
$(0,0)$	$(m_{11},4)$	$(m_{21},10)$	$(m_{31},5)$
$(0,2)$	$(m_{11},2)$	$(m_{21},1),(m_{22},2)$	$(m_{31},2)$
$(1,1)$	$(m_{11},1)$	$(m_{21},2)$	$(m_{31},1)$
$(1,3)$	$(m_{11},3)$	$(m_{21},5),(m_{22},1),$	$(m_{31},3)$
$(2,0)$	$(m_{11},4)$	$(m_{21},5),(m_{22},2)$	$(m_{31},5)$
$(2,2)$	$(m_{11},2)$	$(m_{21},2),(m_{22},1),$	$(m_{31},2)$
$(3,1)$	$(m_{11},1)$	$(m_{21},1)$	$(m_{31},1)$
$(3,3)$	$(m_{11},3)$	$(m_{21},4),(m_{22},1),$	$(m_{31},3)$

TABLE 3.4 – Exemples des chemins pour $(\alpha_{22}, \alpha_{31}) \in \alpha^{(3)}(p_{12}, p_{22}, p_{31})$

Etant données les valeurs de $\alpha^{(3)}(p_{12}, p_{22}, p_{31})$, les configurations parallèles pour l'exécution concurrente de ces trois modes peuvent être déterminées grâce au Lemme 3.8. La Table 3.5 résume toutes les valeurs de δ_{22} et δ_{31} qui constituent une configuration parallèle observable $((m_{12},2),(m_{22},\delta_{22}),(m_{31},\delta_{31}))$.

	$(0,0)$	$(0,2)$	$(1,1)$	$(1,3)$	$(2,0)$	$(2,2)$	$(3,1)$	$(3,3)$
δ_{22}	2	2	4	4	6	6	0	0
δ_{31}	2	6	4	0	2	6	4	0

TABLE 3.5 – Valeurs possibles de δ_{22} et δ_{31} pour les configurations parallèles $((m_{12},2),(m_{22},\delta_{22}),(m_{12},\delta_{31}))$ réellement observables

Cette table ne contient pas les configurations parallèles qui invalident les conditions du Théorème 3 parce que ces configurations sont, en fait, inatteignables et les traces d'exécution des différents modules qui requièrent une unité de temps de processeur sur l'intervalle $\Delta = 2$ ne sont jamais activées simultanément. Par conséquent, la somme de fonctions de demandes maximales pour les traces de cette durée dans les modules différents ne dépasse 2 pour aucune des configurations parallèles. Le système est donc, conformément aux conditions du Théorème 3, ordonnançable.

3.8 Condition d'ordonnançabilité pour plusieurs modules avec ressources allouées par un serveur

Le Théorème 3 est adapté au cas où les ressources (le temps processeur) sont allouées par le biais d'un serveur. Cette allocation s'effectue périodiquement (*Periodic Resource Model* [156]). Les deux conditions du Théorème 3 ainsi que la borne de faisabilité doivent être déterminées en tenant compte du schéma d'allocation.

Un serveur de ressources périodique $\Gamma(\Pi, \Theta)$ délivre Θ unités de ressources par période Π ($\Theta \leq \Pi$). La capacité de ce serveur est égale à $\frac{\Theta}{\Pi}$. La fonction d'approvisionnement minimal $sbf_\Gamma(\Delta)$ (*supply bound function*) décrit la quantité minimale de ressources délivrées dans chaque intervalle de durée Δ. Elle est calculée comme suit :

$$sbf_\Gamma(\Delta) = \left\lfloor \frac{\Delta - (\Pi - \Theta)}{\Pi} \right\rfloor \Theta + \varepsilon_s \tag{3.54}$$

où

$$\varepsilon_s = \max\left(\Delta - 2\,(\Pi - \Theta) - \Pi \left\lfloor \frac{\Delta - (\Pi - \Theta)}{\Pi} \right\rfloor, 0\right)$$

La fonction sbf_Γ est une fonction non décroissante échelonnée. Elle peut être bornée inférieurement par une fonction linéaire. Cette fonction, $lsbf_\Gamma$ (*linear supply bound function*), est donnée par la formule suivante :

$$lsbf_\Gamma(\Delta) = \frac{\Theta}{\Pi}\left(\Delta - 2\,(\Pi - \Theta)\right) \leq sbf_\Pi(\Delta) \tag{3.55}$$

Il est nécessaire d'établir la borne de faisabilité pour un système constitué d'un ensemble de modules $Modules$ dont les ressources (le temps de traitement) sont allouées par un serveur de ressources périodique $\Gamma = (\Pi, \Theta)$. Le raisonnement de la section 3.5.2 est repris en prenant en compte que le temps processeur disponible pour l'exécution de tâches dans tout intervalle Δ ne dépasse pas $lsbf_\Gamma(\Delta)$. Ainsi, l'Equation 3.23 devient :

$$\sum_{M \in Modules} mdbf_M(\Delta) \;>\; lsbf_\Gamma(\Delta) \tag{3.56}$$

En résolvant cette équation la borne de faisabilité ci-dessous est obtenue :

$$\Delta < 2\,\frac{\sum_{M \in Modules} \max_{m \in Modes[M]} (U(m)H[m]) + \frac{\Theta}{\Pi}(\Pi - \Theta)}{\frac{\Theta}{\Pi} - \sum_{M \in Modules} \max_{m \in Modes[M]} U(m)} \tag{3.57}$$

La plus grande valeur de Δ qui satisfait l'Equation 3.57 est $\Delta^\Gamma_{max}(Modules)$.

Le Théorème 3 peut être réécrit.

Théorème 4 (**Version du Théorème 3 en présence d'un serveur de ressources périodique**). *Soit un système exécutant un ensemble $Modules$ de n modules E-TDL. Soit un serveur de ressources périodique $\Gamma = (\Pi, \Theta)$ qui délivre le temps de traitement pour l'exécution des tâches de $Modules$. Le système est ordonnançable sous* EDF *sur un processeur si et seulement si :*

$$\sum_{k=1}^{n} \max_{m \in Modes[M_k]} \{U(m)\} \leq \frac{\Theta}{\Pi}, \tag{3.58}$$

et pour chaque configuration parallèle $\zeta = ((m_1, \delta_1), \ldots, (m_n, \delta_n))$ et chaque intervalle de temps $0 < \Delta \leq \Delta^{\Gamma}_{max}(Modules)$:

$$\sum_{k=1}^{n} \quad maxdf_{M_k}(m_k, \delta_k, \Delta) \leq sbf_{\Gamma}(\Delta) \tag{3.59}$$

où (m_k, δ_k) est un état du module $M_k \in Modules$ dans la configuration parallèle ζ.

Démonstration. La preuve est dérivée de la preuve du Théorème 3. La première condition exprime que le taux d'utilisation ne peut pas être supérieur à la capacité du serveur Γ. La seconde condition vérifie que le temps nécessaire pour le traitement des tâches de $Modules$ dans l'intervalle Δ ne dépasse pas le temps minimal alloué par le serveur Γ dans cet intervalle. □

3.9 Conclusion

La contribution la plus importante présentée dans ce chapitre est d'établir des conditions nécessaires et suffisantes pour l'ordonnançabilité d'un système de plusieurs modules *E-TDL* exécutés sous *EDF* dans un contexte monoprocesseur. Elle repose sur le concept de *fonction de demande* largement appliqué pour évaluer la faisabilité des systèmes utilisant cette politique d'ordonnancement. L'exactitude de ces conditions est due au fait que, dans les modules s'exécutant concurremment, les instants d'activation des tâches et les intervalles entre ces instants peuvent être déterminés par l'analyse développée. Cette analyse s'appuie, dans une certaine mesure, sur les observations qui ont permis à Pellizzoni et Lipari [132, 133] de reproduire les configurations de démarrages dans un ensemble de tâches asynchrone. Elle propose la détermination d'une borne sur la longueur des intervalles qui doivent être vérifiés. Cette borne est obtenue par évaluation de la demande maximale dans les différentes phases d'exécution d'un module et par application du raisonnement proposé par Baruah [24, 25].

Le champ d'application de cette étude ne se réduit pas au seul langage *E-TDL*. Les questions traitées sont en effet communes à la plupart des systèmes dirigés par le temps conçus de manière compositionnelle. Il est alors envisageable que les résultats obtenus dans ce chapitre soient applicables dans le cadre plus large des systèmes à déclenchement temporel.

A ce stade, la question de la mise en œuvre de tous les éléments de l'analyse proposée reste ouverte. Il s'agit particulièrement du calcul de fonction de demande, de l'exploration du graphe de changement de mode et de l'identification des configurations parallèles. Ils permettraient de construire un outil de vérification des systèmes décrits en E-TDL. Le chapitre suivant présente cet outil en accordant une attention particulière aux solutions algorithmiques sur lesquelles il est basé.

Chapitre 4

Un outil pour la vérification d'ordonnançabilité de systèmes décrits en E-TDL

Ce chapitre présente un outil pour l'analyse d'ordonnançabilité d'un système exécutant un ensemble de modules *E-TDL* sous *EDF*. L'outil prend comme paramètre la quantité de ressources disponibles pour l'exécution. Il vérifie si ces ressources sont suffisantes pour que le système soit ordonnançable. Outre une réponse sur l'ordonnançabilité, l'outil évalue aussi les ressources effectivement utilisées pour le traitement des composants d'E-TDL. Ceci permet d'envisager les systèmes d'une manière compositionnelle, les composants E-TDL pouvant partager les ressources de processeur avec d'autres composants.

L'outil est fondé sur les bases théoriques du chapitre précédent. Tout d'abord, il propose un algorithme qui effectue la recherche des instants de débuts des modes dans un graphe de changement de mode. Cela est nécessaire pour établir toutes les configurations parallèles. Ces configurations peuvent être obtenues grâce à une analyse attentive des relations entre instants de débuts des modes dans tous les modules du système. Ensuite, il quantifie la demande maximale de processeur dans tout intervalle. Compte tenu des changements de modes possibles, plusieurs scénarios d'exécution d'un même module doivent être examinés. A partir de tous ces éléments, un test d'ordonnançabilité est établi.

4.1 Introduction

Pour que le test soit efficace, les méthodes qu'il utilise doivent être adaptées à la particularité du problème réellement traité. L'Exemple 3.15 montre que l'analyse exhaustive et exacte, dans certains cas, n'est pas indispensable pour vérifier l'ordonnançabilité dans tout intervalle. Les conditions suffisantes, lorsqu'il est possible de les appliquer, permettent de garantir l'ordonnançabilité plus rapidement. La proposition d'un test qui combine ces

deux types de conditions est formulée dans ce qui suit.

4.1.1 Présentation générale du test

De manière générale, le test d'ordonnançabilité proposé ci-dessous consiste à vérifier pour tout intervalle, dans les limites de la borne de faisabilité, que la quantité de demande nécessaire à l'exécution d'un ensemble de modules E-TDL ne dépasse pas la quantité de traitement disponible (temps de processeur). Ce test repose sur les Théorèmes 2 et 3.
Le Théorème 2 borne supérieurement la quantité de demande :

(a) il fait l'hypothèse que les demandes les plus importantes se produisent en même temps dans les différents modules,

(b) il calcule la demande comme la somme de ces demandes.

Le Théorème 3 estime la quantité de demande de manière exacte :

(a) il considère toutes les configurations parallèles qui peuvent se produire réellement au cours de l'exécution des modules,

(b) il cumule les demandes associées à chacune de ces configurations parallèles.

Le premier calcul est moins complexe que le second puisqu'il effectue une seule opération d'addition des demandes. Le second répète cette opération pour chaque configuration parallèle. C'est pourquoi, on vérifie d'abord les conditions (suffisantes) du Théorème 2. Si ces conditions ne sont pas satisfaites pour un intervalle donné, on vérifie cet intervalle avec les conditions (nécessaires et suffisantes) du Théorème 3.

Le Diagramme 4.1 présente une idée générale du test. Le test commence par vérifier les conditions préalables sur le taux d'utilisation des modules (Les Equations 3.26, 3.52, 3.58). Ensuite, il détermine une borne de faisabilité Δ_{max} (Les Equations 3.25 et 3.57) et procède, dans l'ordre croissant, à la vérification d'ordonnançabilité dans les intervalles Δ. Comme mentionné ci-dessus, il essaye d'abord de vérifier la condition suffisante du Théorème 2. Il calcule donc la somme des bornes des fonction des demandes maximales $mdbf_M(\Delta)$ (l'Equation 3.13) pour tous les modules M dans l'intervalle Δ. Si cette somme dépasse le nombre de ressources alloué dans cet intervalle, $sbf(\Delta)$, il passe à l'évaluation de la condition nécessaire et suffisante du Théorème 3. A cet effet, il doit déterminer toutes les configurations parallèles dans lesquelles les modules peuvent se trouver. Une fois ces configurations obtenues, les sommes de fonctions des demandes maximales (associées à chacune de ces configurations) $maxdf_M(m, \delta, \Delta)$ (l'Equation 3.12) pour tous les modules sont calculées. La plus grande de ces sommes ne doit pas être supérieure à $sbf(\Delta)$. Faute de quoi le système est non ordonnançable et le test s'arrête. L'ordonnançabilité du système est prouvée si le test ne s'arrête dans cet état dans aucun intervalle Δ allant de 1 à Δ_{max}.

4.1.2 Architecture de l'outil

L'analyse fonctionnelle du test fait apparaître les composants suivants qui sont décrits en détails dans la suite de ce chapitre :

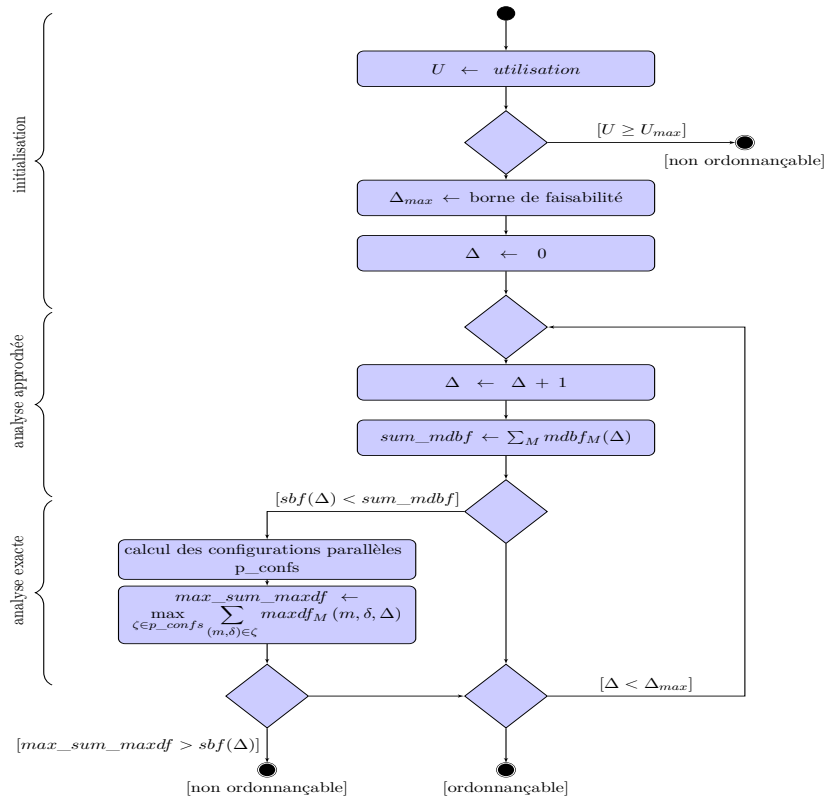


FIGURE 4.1 – Diagramme d'activité du test d'ordonnançabilité pour E-TDL

- **test d'ordonnançabilité** — exécution du test d'ordonnançabilité pour E-TDL (section 4.6)
- **fonctions de demande** — calcul des fonctions de demande (df, $maxdf$) et de la borne de faisabilité (section 4.5)
- **configurations parallèles** — algorithmes de recherche des configurations parallèles d'un module (section 4.4)
- **graphe de changement de mode** — parcours du graphe de changement de mode afin de déterminer les instants de débuts des modes (section 4.3)
- **interface E-TDL** — classes représentant les principales entités du langage E-TDL : tâches, modes, modules (section 4.2)

Le code source de l'outil s'organise autour de cette structure. Chaque fichier fournit les fonctionnalités du composant qu'il met en œuvre. L'outil est écrit en *Python (2.7.6)* et

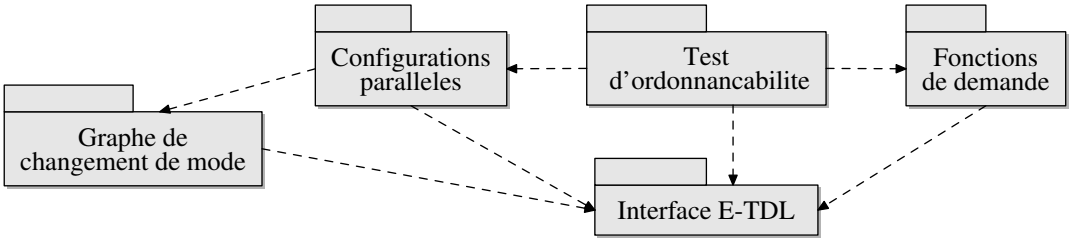


FIGURE 4.2 – Diagramme de paquetages pour l'outil de vérification d'ordonnançabilité pour E-TDL

utilise la bibliothèque *NetworkX* [80]. Ce choix a été dicté par la très bonne efficacité de programmation dans ce langage ainsi que par les nombreux types de base et les extensions dont il est doté. Destiné à s'intégrer au sein d'une chaîne de compilation pour les systèmes à déclenchement temporel, l'outil est conçu sous la forme d'une bibliothèque. Cette bibliothèque fournit les abstractions permettant de décrire un système en E-TDL ainsi que les fonctions qui vérifient son ordonnançabilité. A ce jour, la bibliothèque est constituée d'environ 700 lignes de code (et presque 300 lignes de commentaires). Des exemples de systèmes E-TDL et des jeux de tests visant à évaluer la performance et à donner un aperçu de fonctionnement de chacun des composants ont également été développés.

4.2 E-TDL interface

Un modèle de système s'exprime en E-TDL à l'aide de trois entités principales : le module, le mode, la tâche. La Figure 4.3 montre les classes utilisées pour représenter chacune de ces entités. Elles sont décrites plus précisément ci-dessous.

4.2.1 Classe TDLtask

La classe *TDLtask* est une représentation abstraite de la tâche temps-réel en *E-TDL*. Elle possède les mêmes paramètres que le modèle de cette tâche. Lors de la création d'une instance de classe *TDLtask*, la conformité de ses paramètres avec le modèle de tâche (voir section 3.1) est vérifiée.

4.2.2 Classe TDLmode

La classe *TDLmode* rassemble des caractéristiques et des méthodes relatives au mode en *E-TDL*. De façon générale, un mode exécute des tâches périodiquement. La variable *taskset* d'une instance de cette classe représentant un mode m est la liste des instances de classe *TDLtask* des tâches exécutées par m. La capacité à changer de mode est exprimée

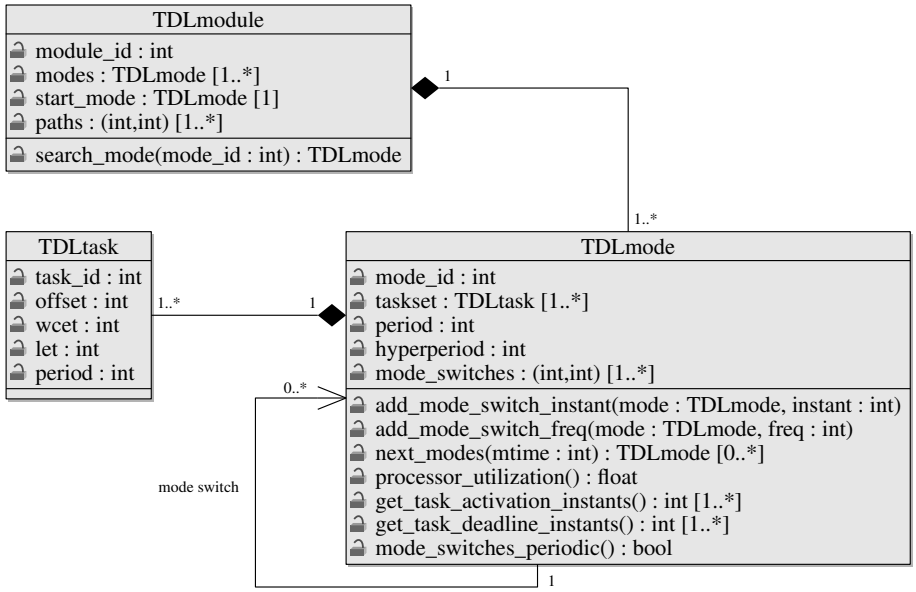


FIGURE 4.3 – Classes pour E-TDL : TDLmodule, TDLmode, TDLtask

par la variable *mode_switches*. Cette variable associe à un instant de mode δ un mode pour lequel une condition de changement est évaluée à δ dans m (plus d'un commutateur peuvent être spécifiés pour un même instant). Les méthodes *add_mode_switch_instant* et *add_mode_switch_freq* permettent de déclarer un commutateur de mode à un ou plusieurs instants d'un mode. La méthode *next_modes* permet de trouver tous les modes dont les commutateurs sont vérifiés à un temps donné du mode. L'exigence de périodicité des instants de vérification des commutateurs de modes est examinée par la méthode *mode_switches_periodic*. Les méthodes *get_task_activation_instants* et *get_task_deadline_instants* retournent des listes qui contiennent respectivement toutes les dates d'activation et toutes les dates d'échéances des tâches d'un mode. Le taux d'utilisation de processeur par un mode est calculé à l'aide de la méthode *processor_utilization*.

4.2.3 Classe TDLmodule

Un module *E-TDL* regroupe des modes et à un instant donné exécute un des ces modes. Ainsi, une instance de la classe *TDLmodule* contient la liste des modes qu'un module peut exécuter. Cette liste est appelée *modes*. La variable *start_mode* désigne le mode par lequel un module commence son exécution. La structure *paths* associe à chaque mode de la liste *modes* l'ensemble des plus grands diviseurs des chaînes qui mènent du mode initial à ce mode. Cette structure est construite par un algorithme présenté dans la section suivante.

4.3 Parcours d'un graphe de changement de mode

Pour déterminer les instants de début d'un mode il est nécessaire de connaître tous les chemins, dans le graphe de changement de mode, qui conduisent du mode initial à ce mode (voir section 3.6.1). Pour cela, il est nécessaire de construire, pour un module, son graphe de changements de mode. Ensuite, il est nécessaire de le parcourir afin de trouver les plus grands communs diviseurs des instants de début de chacun des modes représentés sur ce graphe.

L'objectif est de trouver dans un premier temps et pour un module donné son graphe de changements de mode. Ce graphe, au sens de la Définition 3.1, est constitué par l'ensemble des modes exécutés par le module et par l'ensemble des transitions entre modes qui peuvent survenir pendant son exécution. Un graphe orienté est construit à partir de ces deux ensembles,. Il relie par un arc entre deux sommets s'il existe un commutateur de mode pour les modes représentés par ces sommets. Ensuite, cet arc est étiqueté par la fréquence d'évaluation du commutateur. Rappelons que dans la version actuelle d'E-TDL les conditions des commutateurs de mode sont évaluées périodiquement à des fréquences constantes. La structure du graphe définie tient compte de cette hypothèse et ne serait plus valide si cette dernière était invalidée. L'algorithme ci-dessous produit le graphe de changements de mode G_M en acceptant comme entrée un module M.

Algorithme 1 *Module_to_graphe*(M)

requiert: M est un module E-TDL
assure: $G_M = (V, E)$ est un graphe de changements de mode du module M
1: $V \leftarrow \emptyset$
2: $E \leftarrow \emptyset$
3: **pour tout** $m \in Modes[M]$ **faire**
4: $\quad V \leftarrow V \cup \{m\}$
5: $\quad$ **pour tout** $m' \in \{Modes[M] - \{m\}\}$ **faire**
6: $\quad\quad$ **si** $T_{sw}(m, m') \neq \emptyset$ **alors**
7: $\quad\quad\quad E \leftarrow E \cup \{(m, m', T_{sw}(m, m'))\}$
8: $\quad\quad$ **fin si**
9: $\quad$ **fin pour tout**
10: **fin pour tout**
11: **renvoie** $G_M = (V, E)$

Il existe de manière générale, deux façons d'explorer des graphes : en profondeur (*depth-first search*, *DFS*) ou en largeur (*breadth-first search*, *BFS*). Le parcours en largeur consiste à visiter à partir d'un nœud donné tous ses successeurs et à répéter cette opération pour chacun d'eux tant qu'il existe encore des nœuds non visités. Le parcours en profondeur suit le chemin issu d'un nœud aussi loin que possible et revient au dernier choix fait en prenant une autre direction dès que tous les successeurs du nœud courant sont déjà visités ou n'existent pas. L'algorithme d'exploration du graphe de changement de mode proposé ci-dessous est basé sur cette dernière approche (*DFS*). Il existe deux versions de mise en œuvre de *DFS* : récursive [158] et non-récursive avec une pile [96]. C'est la version avec

pile qui est utilisée.

La pile est une liste linéaire pour laquelle toutes les opérations d'insertion ou de suppression ne peuvent être réalisées qu'à son sommet [101]. Le dernier élément inséré sur la pile est également le premier élément à en être retiré. Dans l'algorithme qui va suivre, pour une structure de données appelée *Pile* :

Pile.empiler(x) ajoute l'élément x au sommet de la *Pile*,

Pile.dépiler() retire de *Pile* le dernier élément qui y a été empilé,

Pile.sommet() retourne le dernier élément empilé sur la *Pile*.

L'algorithme proposé n'a pas seulement pour but de visiter tous les nœuds d'un graphe de changement de mode d'un module. Son premier objectif est de déterminer les plus grands communs diviseurs de tous les instants de début de modes représentés par ce graphe. Dans la version classique du *DFS*, un nœud qui est visité est marqué afin de le classer dans la catégorie des nœuds déjà visités. Dans le cas présent, au lieu d'un simple marquage, les plus grands communs diviseurs des instants de début des modes qui correspondent aux nœuds visités par l'algorithme sont calculés. Plusieurs chemins peuvent mener à un nœud et par conséquent plusieurs valeurs peuvent être obtenues en prenant ces différents chemins.

Remarque 4.1. *Si $P_{prec}(m_k)$ est l'ensemble de toutes les chaînes qui mènent depuis le mode initial d'un module M jusqu'à un des modes qui précèdent directement un mode $m_k \in Modes[M]$ dans le graphe de changement de mode du module M, les instants de début de mode m_j peuvent s'exprimer comme :*

$$start(m_k) \in \{\, n\, pgcd\,(p_j \cup \{T_{sw}(m_j, m_k), T[m_k]\}) \mid p_j \in P_{prec}(m_k),\, m_j = tail(p_j),\, n \in \mathbb{N}_+ \,\} \tag{4.1}$$

Démonstration. Un mode m_j précède directement le mode m_k dans le graphe de changement de mode. Les instants de début de mode m_k peuvent se produire $n_1 \cdot T_{sw}(m_j, m_k) + n_2 \cdot T[m_k]$ après les instants de début de mode m_k où $n_1, n_2 \in \mathbb{N}_+$.

$$start(m_k) = \{\, t_{start}(m_j) + n_1 \cdot T_{sw}(m_j, m_k) + n_2 \cdot T[m_k] \mid n_1, n_2 \in \mathbb{N}_+ \,\}$$

D'après la Formule 3.34 et en exprimant la combinaison linéaire $n_1 \cdot T_{sw}(m_j, m_k) + n_2 \cdot T[m_k]$ comme $n' \cdot pgcd(T_{sw}(m_j, m_k), T[m_k])$ on obtient :

$$start(m_k) \in \{\, n\, pgcd(p_j \cup \{T_{sw}(m_j, m_k), T[m_k])\} \mid n \in \mathbb{N}_+, p_j \in P_j \,\}$$

où P_j est l'ensemble de toutes les chaînes qui mènent depuis le mode initial du module M jusqu'au mode m_j. On étend cette relation sur tous les modes qui précèdent le mode m_k dans le graphe de changement de mode du module M. Tous ces modes sont atteignables par l'ensemble de chaînes $P_{prec}(m_k)$. □

A chaque visite d'un nœud, dénoté par *act*, les ensembles des plus grands communs diviseurs *PGCD[s]* de chacun de ses successeurs s ($s \in voisins(act)$) sont mis à jour à l'aide de la relation suivante :

$$PGCD[s] = PGCD[s] \cup \{\, pgcd(T[s], T_{sw}(act, s), \pi) \mid \pi \in PGCD[act] \,\} \tag{4.2}$$

Ensuite, on vérifie que les valeurs ajoutées à l'ensemble $PGCD[s]$ ne sont pas des multiples de valeurs déjà présentes dans $PGCD[s]$ (il est inutile de retenir qu'un mode peut commencer aux instants $8 \cdot n$ si on sait déjà qu'il peut commencer aux instants $4 \cdot n$). Si ces opérations modifient l'ensemble $PGCD[s]$ pour au moins un nœud s parmi tous les successeurs *d'act*, l'algorithme procède à la visite de s et répète la procédure pour tous les successeurs de celui-ci. Dans le cas contraire, il revient au nœud qui a été visité avant *act* et essaie de visiter les autres successeurs. L'algorithme garde la trace de son chemin en empilant les nœuds visités et en dépilant ceux desquels il se retire.

Exemple 4.1. *Soit un module E-TDL qui exécute les modes suivants :* $Modes[M] = \{m_a, m_b, m_c, m_d, m_e\}$ *le mode* m_a *étant le mode initial. Les commutateurs de mode sont évalués dans les modes avec les fréquences suivantes :* $T_{sw}(m_a, m_b) = 10$, $T_{sw}(m_a, m_e) = 3$, $T_{sw}(m_b, m_c) = 15$, $T_{sw}(m_b, m_d) = 20$, $T_{sw}(m_d, m_b) = 6$ *et* $T_{sw}(m_d, m_e) = 10$. *Les périodes de mode sont :* $T[m_a] = 30$, $T[m_b] = 60$, $T[m_c] = 300$, $T[m_d] = 30$ *et* $T[m_e] = 120$. *La Figure 4.4 illustre le graphe de changement de mode du module* M. *Le fonctionnement de l'Algorithme 2 pour le parcours de ce graphe de changement de mode est retracé étape par étape dans la Table 4.1. La première colonne reflète la composition de la pile tandis*

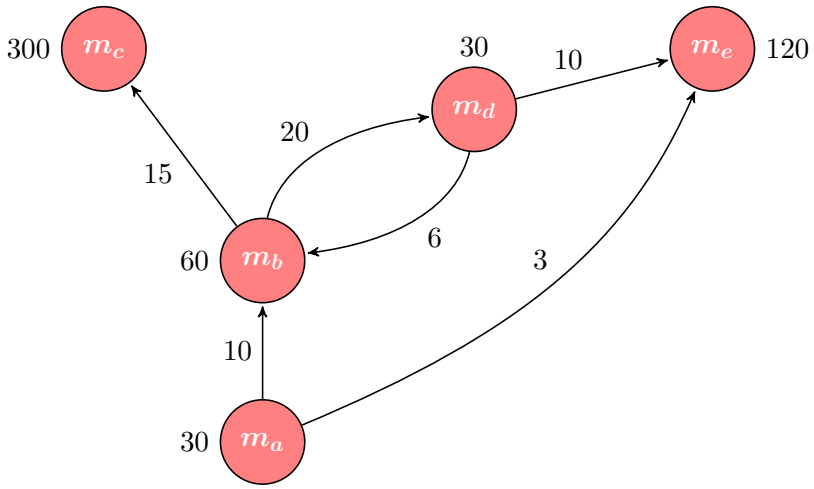


FIGURE 4.4 – Graphe de changement de mode du module M de l'Exemple 4.1

que les colonnes suivantes illustrent les valeurs des plus grands communs diviseurs des

Algorithme 2 $DFSgcd(G_M)$

requiert: G_M est le graphe de changements de mode d'un module M
assure: $\forall m \in Modes[M] : PGCD[m] = \{pgcd(p_m) \mid head(p_m) = Init[M],\ tail(p_m) = m\}$

1: $PGCD[Init[M]] \leftarrow T[Init[M]]$
2: $Pile \leftarrow \emptyset$
3: $Pile.empiler(Init[M])$
4: début :
5: **tant que** $Pile \neq \emptyset$ **faire**
6: $act \leftarrow Pile.sommet()$
7: **pour tout** $s \in voisins(act)$ **faire**
8: **si** VISITER$(G_M, act, s, PGCD)$ **alors**
9: $Pile.empiler(s)$
10: **saut** début
11: **fin si**
12: **fin pour tout**
13: $Pile.dépiler()$
14: **fin tant que**
15: **renvoie** $PGCD$
16:
17: **function** VISITER$(G_M, act, s, PGCD)$
18: $PGCD' \leftarrow \{pgcd(T[s], T_{sw}(act, s), \pi) \mid \pi \in PGCD[act]\}$
19: $PGCD' \leftarrow PGCD' \cup PGCD[s]$
20: **si** $|PGCD| > 1$ **alors**
21: **pour tout** $\pi \in PGCD'$ **faire**
22: $PGCD' \leftarrow PGCD' - \{\pi\}$
23: **si** $\forall \pi' \in PGCD'\ \nexists n \in \mathbb{N} : n \cdot \pi' = \pi$ **alors**
24: $PGCD' \leftarrow PGCD' \cup \{\pi\}$
25: **fin si**
26: **fin pour tout**
27: **fin si**
28: **si** $PGCD' \neq PGCD[s]$ **alors**
29: $PGCD[s] \leftarrow PGCD'$
30: **renvoie** $vrai$
31: **sinon**
32: **renvoie** $faux$
33: **fin si**
34: **fin function**

instants de début de mode, $PGCD$, au fur et à mesure du parcours effectué par l'algorithme. L'algorithme commence l'exploration du graphe par le nœud a qui correspond au mode initial m_a (la première ligne dans le tableau) et visite les nœuds suivants en essayant toujours d'aller aussi loin que possible (les lignes suivantes).
Le graphe possède un cycle ($m_b - m_d$) et toutes les valeurs de $PGCD$ pour les nœuds de ce

pile	$PGCD[m_a]$	$PGCD[m_b]$	$PGCD[m_c]$	$PGCD[m_d]$	$PGCD[m_e]$
a	30	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
a,b	30	10	$\emptyset$	$\emptyset$	$\emptyset$
a,b,c	30	10	5	$\emptyset$	$\emptyset$
a,b	30	10	5	$\emptyset$	$\emptyset$
a,b,d	30	10	5	10	$\emptyset$
a,b,d,b	30	2	5	10	$\emptyset$
a,b,d,b,c	30	2	1	10	$\emptyset$
a,b,d,b	30	2	1	10	$\emptyset$
a,b,d,b,d	30	2	1	2	$\emptyset$
a,b,d,b,d,e	30	2	1	2	2
a,b,d,b,d	30	2	1	2	2
a,b,d,b	30	2	1	2	2
a,b,d	30	2	1	2	2
a,b	30	2	1	2	2
a	30	2	1	2	2
a,e	30	2	1	2	2, 3
a	30	2	1	2	2, 3

TABLE 4.1 – Fonctionnement de l'Algorithme 2 pour le module M de l'Exemple 4.1.

cycle sont identiques. Cette observation peut aider à réduire la complexité de l'algorithme.

Remarque 4.2. *Si dans le graphe de changement de mode d'un module M il existe un cycle formé par une chaîne p_c qui traverse les modes $m_x \in Modes[M]$ et $m_y \in Modes[M]$, les plus grands communs diviseurs des instants de début de ces deux modes sont identiques.*

Démonstration. Soient P_x et P_y des ensembles de chaînes qui permettent d'accéder aux modes m_x et m_y depuis le mode initial $Init[M]$. Il est évident que tout chemin qui suit la chaîne $p_x \in P_x$ peut être prolongé depuis m_x en prenant le cycle p_c et parvenir à nouveau au mode m_x :

$$\{\, p_x \cup p_c \,|\, p_x \in P_x \,\} \subseteq P_x$$

Le plus grand commun diviseur d'une chaîne p_x est multiple du plus grand commun diviseur

d'une chaîne qui est composée des chaînes p_x et p_c :

$$\forall p_x \in P_x : \{\, n\, pgcd(p_x) \,|\, n \in \mathbb{N}_+ \,\} \subseteq \{\, n\, pgcd(p_x \cup p_c) \,|\, n \in \mathbb{N}_+ \}$$

Alors, dans l'expression 3.34 on peut, sans perte de généralité, considérer uniquement les chaînes qui traversent le cycle p_c :

$$t_{start}(m_x) \in \bigcup_{p_x \in P_x} \{\, n\, pgcd(p_x \cup p_c) \,|\, n \in \mathbb{N} \,\}$$

Le même raisonnement est mené pour le mode m_y :

$$t_{start}(m_y) \in \bigcup_{p_y \in P_y} \{\, n\, pgcd(p_y \cup p_c) \,|\, n \in \mathbb{N} \,\}$$

Toute chaîne $p_x \cup p_c$ passe par le mode m_y et inversement toute chaîne $p_y \cup p_c$ passe par le mode m_x :

$$\bigcup_{p_x \in P_x} \{\, p_x \cup p_c \,\} \subseteq \bigcup_{p_y \in P_y} \{\, p_y \cup p_c \,\} \quad et \quad \bigcup_{p_y \in P_y} \{\, p_y \cup p_c \,\} \subseteq \bigcup_{p_x \in P_x} \{\, p_x \cup p_c \,\}$$

Alors les deux ensembles $\bigcup_{p_x \in P_x} \{\, p_x \cup p_c \,\}$ et $\bigcup_{p_y \in P_y} \{\, p_y \cup p_c \,\}$ sont égaux :

$$\bigcup_{p_x \in P_x} \{\, p_x \cup p_c \,\} \equiv \bigcup_{p_y \in P_y} \{\, p_y \cup p_c \,\}$$

□

Les modes qui font partie d'un même cycle dans le graphe de changement de mode peuvent commencer leurs exécutions à des instants multiples des mêmes coefficients. Il est donc inutile de parcourir à chaque fois tous les modes de ce cycle si cela conduit aux mêmes résultats pour tous ces modes. En effectuant une condensation du graphe de changement de mode en remplaçant tous les nœuds d'un cycle par un seul nœud le nombre des nœuds visités par l'algorithme peut être réduit. La détection des cycles peut se faire en observant la pile. Si un nœud est empilé pour la deuxième fois, tous les nœuds empilés entre les deux occurrences de ce nœud forment un cycle. Cette nouvelle version de l'algorithme a été comparée avec celle proposée auparavant. En général, les graphes de changement de mode utilisés dans les tests ne comportaient pas plus d'une dizaine de modes. Ce nombre était trop faible pour pouvoir observer une amélioration significative du fonctionnement de l'algorithme. Pour cette raison, et sans vouloir rendre la solution plus complexe qu'elle ne l'est, c'est la version initiale qui est utilisée. Néanmoins, la version améliorée de l'algorithme est conservée dans les sources et peut remplacer la version actuelle si cela apparaît utile.

La mise en œuvre de l'algorithme nécessite des structures de données qui permettent de représenter les graphes et d'effectuer des opérations élémentaires sur ces derniers. Pour cela, il est fait appel à *NetworkX* [80], une bibliothèque de Python, pour l'étude des graphes et des réseaux. Cette bibliothèque contient de nombreuses classes et méthodes pour construire, manipuler et analyser les graphes orientés et non-orientés. La librairie est distribuée librement sous la licence *BSD*.

4.4 Configurations parallèles

Ce qui suit montre comment déterminer les configurations parallèles dont la connaissance est essentielle dans l'analyse exacte d'ordonnançabilité d'un système modélisé en *E-TDL*. Le premier pas vers la résolution de ce problème consiste à trouver l'ensemble des configurations de démarrage des modes s'exécutant en même temps dans les différents modules du système. En s'appuyant sur les méthodes d'exploration des graphes de changement de mode présentées dans la section précédente il est possible de trouver les instants de démarrage des modes. Les configurations de démarrage sont obtenues en situant ces instants les uns par rapport aux autres. Ensuite, les configurations de démarrage de modes sont transformées en configurations parallèles induites par la détermination du positionnement des démarrages de modes. La dernière étape consiste à proposer une méthode qui permette d'obtenir toutes les configurations parallèles coïncidant avec un état de module librement choisi.

4.4.1 Configurations de démarrage de modes

Les Lemmes 3.5 et 3.6 caractérisent les relations que les instants de démarrage de modes s'exécutant concurremment dans les différents modules doivent satisfaire. Soient les modes $m_1, \ldots, m_n$ qui s'exécutent en même temps et sont atteints, respectivement, par les chaînes $p_1, \ldots, p_n$. Soit une configuration de démarrage des modes $m_1, \ldots, m_{k-1}$, constituée des coefficients $(\alpha_2, \ldots, \alpha_{k-1}) \in \alpha^{(k-1)}(p_1, \ldots, p_{k-1})$, connue et telle que $k < n$. Il faut maintenant déterminer les points de démarrage du mode m_k en accord avec les points de démarrage des modes $m_1, \ldots, m_{k-1}$ définis dans cette configuration. Il est possible de retrouver, avec le Lemme 3.6, tout coefficient α_k pour lequel il existe une configuration de démarrage des modes $m_1, \ldots, m_{k-1}, m_k$ formée comme $(\alpha_2, \ldots, \alpha_{k-1}, \alpha_k)$. En d'autres termes, le Lemme 3.6 permet d'obtenir l'ensemble $\alpha^{(k)}(p_1, \ldots, p_{k-1}, p_k)$ de toutes les configurations de démarrage des modes $m_1, \ldots, m_{k-1}, m_k$ à partir de l'ensemble $\alpha^{(k-1)}(p_1, \ldots, p_{k-1})$.

$$\alpha^{(k-1)}(p_1, \ldots, p_{k-1}) \rightarrow Lemme\,3.6 \rightarrow \alpha^{(k)}(p_1, \ldots, p_k)$$

Ainsi, il est possible de définir une procédure itérative qui, en commençant avec un ensemble de configurations de démarrage pour les deux premiers modes, $\alpha^{(2)}(p_1, p_2)$, ajoute, à chaque étape, un nouveau mode et évalue les configurations de démarrage pour ce nouvel ensemble de modes. La procédure se termine lorsque tous les n modes sont examinés et l'ensemble $\alpha^{(n)}(p_1, \ldots, p_n)$ de leurs configurations de démarrage trouvé.

Dans l'Algorithme 3, cette approche est utilisée pour trouver l'ensemble $\alpha^{(n)}(p_1, \ldots, p_n)$ de toutes les configurations de démarrage des n modes : $m_1, \ldots, m_n$. L'algorithme prend en entrée les périodes des modes et les chaînes qui conduisent à leur exécution. Il trouve tout d'abord un ensemble $\alpha^{(2)}(p_1, p_2)$ de toutes les configurations de démarrage pour les deux premiers modes. Ensuite, il appelle, à chaque itération k $(3 \leq k \leq n)$, la fonction

$find\_new\_start\_confs$ pour toute une configuration $(\alpha_2, \ldots, \alpha_{k-1})$ de démarrage de modes dans l'ensemble $\alpha^{(k-1)}(p_1, \ldots, p_{k-1})$ et une chaîne p_k. Cette fonction retourne un ensemble de configurations $(\alpha_2, \ldots, \alpha_{k-1}, \alpha_k)$ de démarrage de k modes obtenues en appliquant le Lemme 3.6. L'ensemble $\alpha^{(k)}(p_1, \ldots, p_k)$ de toutes les configurations de démarrage des modes $m_1, \ldots, m_k$ est constitué en cumulant les résultats de toutes les invocations de la fonction $find\_new\_start\_confs$ pour chaque configuration $(\alpha_2, \ldots, \alpha_{k-1})$ de démarrage de modes dans l'ensemble $\alpha^{(k-1)}(p_1, \ldots, p_{k-1})$.

Algorithme 3 $Find\_all\_start\_confs((p_1, \ldots, p_n), (T[m_1], \ldots, T[m_n]))$

requiert: $\forall 1 \leq j \leq n : p_j$ est une chaîne de $Init[M_j]$ à m_j et $m_j \in Modes[M_j]$

assure: $\alpha^{(n)}(p_1, \ldots, p_n)$ est l'ensemble de toutes les configurations $(\alpha_2, \ldots, \alpha_n)$ de démarrage des modes $m_1, \ldots, m_n$ si ceux-ci sont atteints par les chaînes $p_1, \ldots, p_n$

function FINDALLSTARTCONFS($(p_1, \ldots, p_n), (T[m_1], \ldots, T[m_n])$)
    $\alpha^{(2)}(p_1, p_2) \leftarrow$FIND_NEW_START_CONFS$((p_1, p_2), T[m_2], \emptyset)$
    pour tout $k \leftarrow 3, n$ **faire**
        $\alpha^{(k)}(p_1, \ldots, p_k) \leftarrow \emptyset$
        pour tout $(\alpha_2, \ldots, \alpha_{k-1}) \in \alpha^{(k-1)}(p_1, \ldots, p_{k-1})$ **faire**
            $\alpha^{(k)}(p_1, \ldots, p_k) \leftarrow$
                $\alpha^{(k)}(p_1, \ldots, p_k) \cup$ FIND_NEW_START_CONFS$((p_1, \ldots, p_k), T[m_k], (\alpha_2, \ldots, \alpha_{k-1}))$
        fin pour tout
    fin pour tout
    renvoie $\alpha^{(n)}(p_1, \ldots, p_n)$
fin function

function FIND_NEW_START_CONFS($(p_1, \ldots, p_k), T[m_k], (\alpha_2, \ldots, \alpha_{k-1})$)
    $nouvelles\_configurations \leftarrow \emptyset$
    $rest \leftarrow 0$
    pour tout $\alpha_k : 0 \leq \alpha_k * pgcd(p_1 \cup p_k) < T[m_k]$ **faire**
        pour tout $j \leftarrow 2, k-1$ **faire**
            $diff \leftarrow \alpha_k * pgcd(p_1 \cup p_k) - \alpha_j * pgcd(p_1 \cup p_j)$
            $rest \leftarrow diff \bmod pgcd(p_j \cup p_k)$
            si $rest \neq 0$ **alors**
                break
            fin si
        fin pour tout
        si $rest = 0$ **alors**
            $nouvelles\_configurations \leftarrow nouvelles\_configurations \cup (\alpha_2, \ldots, \alpha_{k-1}, \alpha_k)$
        fin si
    fin pour tout
    renvoie $nouvelles\_configurations$
fin function

4.4.2 Modes synchrones

Dans la section 3.6.4, le cas des modes dont les instants de démarrage peuvent coïncider de façon arbitraire est évoqué. Les démarrages de ces modes peuvent être positionnés librement dans leurs configurations de démarrage sans vérifier les conditions des Lemmes 3.5 et 3.6. Cela est possible si tous les plus grands communs diviseurs de leurs chaînes sont premiers entre eux (Lemme 3.10). Ces modes peuvent être exclus de l'analyse exacte en les marquant comme modes du cas synchrone. Pour ce faire, l'Algorithme 4 vérifie les chaînes de tous les modes parallèles.

Algorithme 4 $Modes\_synchrones(p_1, \ldots, p_n)$

requiert: $p_1, \ldots, p_n$ sont des chaînes de différents graphes de changements de mode
1: **function** MODES_SYNCHRONES($p_1, \ldots, p_n$)
2: **pour tout** $i \leftarrow 1, n-1$ **faire**
3: **pour tout** $j \leftarrow i+1, n$ **faire**
4: **si** $pgcd(p_i \cup p_j) \neq 1$ **alors**
5: **renvoie** faux
6: **fin si**
7: **fin pour tout**
8: **fin pour tout**
9: **renvoie** vrai
10: **fin function**

4.4.3 Décalages entre débuts de modes concurrents

Les algorithmes précédents permettent de définir une procédure qui détermine les temps de décalages entre activations de modes qui s'exécutent parallèlement. L'Algorithme 5 décrit les étapes nécessaires pour obtenir ces décalages pour toutes les combinaisons de modes concurrents que le système peut exécuter.

Pour chaque configuration des modes $(m_1, \ldots, m_n) \in p\_modes$ pouvant s'exécuter en parallèle, toutes les combinaisons possibles de chaînes conduisant à l'exécution de ces modes $p\_paths[(m_1, \ldots, m_n)]$ sont déterminées. Compte tenu du fait que chaque mode peut être atteint par différentes chaînes, il peut exister plusieurs combinaisons pour une configuration de modes donnée. Si des chaînes de l'ensemble $p\_paths[(m_1, \ldots, m_n)]$ ont leurs plus grands communs diviseurs premiers entre eux, la configuration parallèle des modes $m_1, \ldots, m_n$ est classée dans l'ensemble des modes synchrones *syncs*. Dans le cas contraire, une analyse précise des instants de décalages est menée. Pour chaque combinaison possible de chaînes $(p_1, \ldots, p_n) \in p\_paths[(m_1, \ldots, m_n)]$ l'ensemble des $(n-1)$-uplets $\alpha^{(n)}(p_1, \ldots, p_n)$ de coefficients décrivant les configurations de démarrage des modes $m_1, \ldots, m_n$ est évalué. Les coefficients de chacun de $(n-1)$-uplets dénotant une configuration de démarrage particulière sont utilisés pour calculer en unités de temps les instants de démarrage de cette configuration. Le résultat de cette opération est ajouté à l'ensemble $offsets[(m_1, \ldots, m_n)]$. Les éléments de cet ensemble sont des (n-1)-uplet de la forme : $(\delta_2, \ldots, \delta_n)$ où δ_i est un

intervalle de temps entre l'activation la plus récente du mode m_i et l'activation la plus récente du mode m_1. Le temps de décalage du premier mode est toujours égal à 0.

Algorithme 5 *Get_offsets*($Modules$)

requiert: $Modules = \{M_1, \ldots, M_n\}$ est un ensemble de n modules tel que $n > 1$
assure: $offsets$ est la liste de tous les décalages entre démarrages de modes des $Modules$
assure: $syncs$ est un ensemble de modes parallèles dans le cas synchrone pour $Modules$

1: $p\_modes \leftarrow \{ (m_1, \ldots, m_n) \,|\, \forall 1 \leq i \leq n : m_i \in Modes[M_i] \}$
2: $p\_paths \leftarrow [\,]$
3: **pour tout** $(m_1, \ldots, m_n) \in p\_modes$ **faire**
4: $\quad p\_paths[(m_1, \ldots, m_n)] \leftarrow \{ (p_1, \ldots, p_n) \,|\, \forall 1 \leq i \leq n : tail(p_i) = m_i,\ head(p_i) = Init[M_i] \}$
5: **fin pour tout**
6:
7: $offsets \leftarrow [\,]$
8: $syncs \leftarrow \emptyset$
9: **pour tout** $(m_1, \ldots, m_n) \in p\_modes$ **faire**
10: $\quad$ **si** $\exists (p_1, \ldots, p_n) \in p\_paths[(m_1, \ldots, m_n)]$: MODES_SYNCHRONES$(p_1, \ldots, p_n)$=vrai **alors**
11: $\quad\quad syncs \leftarrow syncs \cup \{ (m_1, \ldots, m_n) \}$
12: $\quad\quad$ **continue**
13: $\quad$ **fin si**
14: $\quad offsets[(m_1, \ldots, m_n)] \leftarrow \emptyset$
15: $\quad$ **pour tout** $(p_1, \ldots, p_n) \in p\_paths[(m_1, \ldots, m_n)]$ **faire**
16: $\quad\quad \alpha^{(n)}(p_1, \ldots, p_n) \leftarrow$FINDALLSTARTCONFS$((p_1, \ldots, p_n), (T[m_1], \ldots, T[m_n]))$
17: $\quad\quad offsets[(m_1, \ldots, m_n)] \leftarrow$
$\quad\quad\quad\quad offsets[(m_1, \ldots, m_n)] \cup$ CALCULATE_OFFSETS$((p_1, \ldots, p_n), \alpha^{(n)}(p_1, \ldots, p_n))$
18: $\quad$ **fin pour tout**
19: **fin pour tout**
20: **renvoie** $offsets$, $syncs$

21: **function** CALCULATE_OFFSETS$((p_1, \ldots, p_n), \alpha^{(n)}(p_1, \ldots, p_n))$
22: $\quad offsets \leftarrow \emptyset$
23: $\quad$ **pour tout** $(\alpha_2, \ldots, \alpha_n) \in \alpha^{(n)}(p_1, \ldots, p_n)$ **faire**
24: $\quad\quad offsets \leftarrow offsets \cup \{ (\alpha_2 * pgcd(p_1 \cup p_2), \alpha_3 * pgcd(p_1 \cup p_3), \ldots, \alpha_n * pgcd(p_1 \cup p_n)) \}$
25: $\quad$ **fin pour tout**
26: $\quad$ **renvoie** $offsets$
27: **fin function**

4.4.4 Recherche de configurations parallèles

On ne s'intéresse qu'aux configurations parallèles qui correspondent aux dates d'activation d'une tâche et, par conséquent, aux démarrages des périodes d'activité de processeur. L'Algorithme 5 calcule les décalages entre débuts des modes concurrents. Ces décalages peuvent être considérés comme les configurations parallèles relatives à l'instant d'activation du mode du premier module de ces configurations : $((m_1, \mathbf{0}), \ldots, (m_n, \delta_n))$. Néanmoins, il est important de pouvoir déterminer d'autres configurations, où le temps de ce mode n'est pas forcément nul. L'Algorithme 6 transforme les décalages entre débuts des modes concurrents obtenus par l'Algorithme 5 en configurations parallèles où le temps dans

le mode d'un des états de module a une valeur donnée. Le fonctionnement de l'algorithme s'appuie sur les résultats du Lemme 3.49. L'algorithme accepte en entrée un ensemble de décalages $offsets$ et un état de module (m_i, δ_i) pour lequel il évalue toutes les configurations parallèles dans lesquelles cet état est présent. Pour cela, pour chaque configuration de décalages d'entrée, il calcule l'intervalle Δ_i au but duquel le système, s'il continue son exécution à partir de l'état de module (m_i, δ'_i) associé à cette configuration de décalages, parviendra à l'état de module (m_i, δ_i). L'algorithme retourne l'ensemble p_confs de toutes les configurations parallèles ζ obtenues ainsi.

Algorithme 6 $shift((m_i, \delta_i), offsets)$

requiert: $0 \leq \delta_i < T[m_i]$ est un temps de mode $m_i \in Modes[M_i]$
requiert: $offsets$ est un résultat de l'Algorithme 5 pour $Modules = \{M_1, \ldots, M_n\}$ et $i \leq n$
assure: p_confs est un ensemble de configurations parallèles ζ telles que $(m_i, \delta_i) \in \zeta$

1: **function** SHIFT$((m_i, \delta_i), offsets)$
2: $p\_confs \leftarrow \emptyset$
3: $p\_modes \leftarrow \{ (m_1, \ldots, m_i, \ldots, m_n) \,|\, \forall 1 \leq j \leq n, j \neq i : m_j \in Modes[M_j] \}$
4: **pour tout** $(m_1, \ldots, m_n) \in p\_modes$ **faire**
5: **pour tout** $(\delta'_2, \ldots, \delta'_n) \in offsets[(m_1, \ldots, m_n)]$ **faire**
6: $\delta'_1 \leftarrow 0$
7: $\Delta_i \leftarrow (\delta_i - \delta'_i) \bmod T[m_i]$
8: $\zeta \leftarrow ((m_1, \delta_1), \ldots, (m_n, \delta_n)) \,|\, \forall 1 \leq j \leq n : \delta_j \leftarrow (\delta'_j + \Delta_i) \bmod T[m_j]$
9: $p\_confs \leftarrow p\_confs \cup \zeta$
10: **fin pour tout**
11: **fin pour tout**
12: **renvoie** p_confs
13: **fin function**

4.5 Calcul des fonctions de demande

Cette section propose des méthodes pour le calcul de la demande maximale de processeur. Elle définit d'abord des structures de données qui reflètent, compte tenu des changements de mode possibles, les états consécutifs d'un module et les charges qui peuvent y être associées. Elle montre ensuite comment projeter un module donné sur ces structures afin de les explorer et de trouver la valeur de la fonction de demande maximale à partir d'un état de ce module. Enfin, les algorithmes permettant d'effectuer cette recherche sont décrits.

4.5.1 Structures de données d'implantation

L'espace de recherche exploré lors du calcul de la demande maximale est d'abord défini. Pour cela il est nécessaire de se doter d'une abstraction permettant de :

(a) représenter tout état de module :
L'état d'un module est défini par le mode et le temps passé dans le mode que le

module exécute à un instant donné. Pour vérifier la condition 3.53 du Théorème 3 il est nécessaire d'examiner, pour tout intervalle Δ, les traces d'exécution démarrant à partir de tout état du module. Certaines tâches, démarrant dans un état donné du module, peuvent initialement ne solliciter aucune activité du système pendant une certaine durée. Ce temps d'inactivité dure jusqu'à la prochaine date d'activation d'une tâche ou d'un nouveau mode. Du point de vue de l'étude de la fonction de demande, il est alors suffisant de considérer uniquement les états de module dans lesquels une nouvelle tâche ou nouveau mode sont démarrés :

$$\{\,(m,\delta)\,|\,m \in M.modes,\ \delta \in \{\,n\,H[m] \cup m.task\_activations\,\}\,\} \\ \text{et } n = 0 \ldots T[m]/H[m]$$

(b) quantifier la charge du processeur dans tout intervalle :
L'objectif est d'évaluer la fonction de demande maximale $maxdf_M(m,\delta,\Delta)$ (Définition 3.10) pour tout intervalle Δ et tout état (m,δ) d'un module M. Toute trace d'exécution déclenchée à l'état (m,δ) se déroule toujours de la même manière sur la durée δ_H ($\delta_H = \delta + H[m] - \delta \bmod H[m]$) qui sépare cet état de la prochaine hyperpériode car aucun changement de mode n'est possible avant. De plus, toutes les tâches doivent être terminées avant δ_H. Il s'en suit que :

$$maxdf_M(m,\delta,\Delta) = \begin{cases} df_M(m,\delta,\delta_H) + \max\limits_{(m',\delta')} \{\, maxdf_M(m',\delta',\Delta') \,\} & \text{si } \Delta > \delta_H - \delta \\ df_M(m,\delta,\delta+\Delta) & \text{sinon} \end{cases} \tag{4.3}$$

où (m',δ') dénote un état pouvant suivre immédiatement l'état (m,δ_H) et $\Delta' = \Delta - (\delta_H - \delta)$. Il est donc possible d'évaluer la fonction de demande maximale en appliquant une approche récursive. Pour cela, les valeurs de la fonction de demande dans chaque mode du module M doivent être connues. Comme montré en section 4.5.3.2, afin de réduire la complexité du calcul, les valeurs de fonction de demande maximale déjà évaluées sont conservées en mémoire et réutilisées ensuite lors des prochaines évaluations de cette fonction.

(c) exprimer les liens entre états successifs d'un module :
La méthode du calcul de la fonction de demande maximale proposée nécessite de savoir quels sont les états vers lesquels une trace peut évoluer à l'hyperpériode. Un état (m',δ') survenant immédiatement après l'état (m,δ_H) correspond soit au début d'un des modes pour lesquels une condition de changement de mode est vérifiée dans l'état (m,δ_H), soit à l'état suivant (m,δ_H+1) dans le même mode :

$$(m',\delta') \in \{\,(m',0)\,|\,m' \in prochains\_modes(m,\delta_H)\,\} \cup \{\,(m,\delta_H+1)\ si\ \delta_H \neq T[m]\,\} \tag{4.4}$$

Pour répondre à ces besoins, un état de module est représenté par une instance de la classe *ModuleState*. Celle-ci est détaillée sur la Figure 4.5 et dans la Table 4.2. La classe

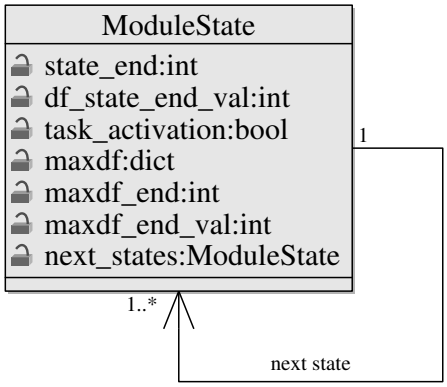


FIGURE 4.5 – Classe *ModuleState*

ModuleState contient plusieurs informations sur l'état de module (m, δ). Son premier paramètre, $state\_end = \delta_H - \delta$, désigne l'intervalle de temps séparant le temps de mode δ et la prochaine hyperpériode δ_H. La valeur de la fonction de demande sur cet intervalle, $df_M(m, \delta, \delta_H)$, est donnée par $df\_state\_end\_val$. Si dans cet état une ou plusieurs tâches sont activées, la valeur de l'attribut $task\_activations$ est mise à vrai. Le paramètre suivant, $maxdf$, est un dictionnaire qui contient toutes les valeurs de la fonction de demande maximale $maxdf_M(m, \delta, \Delta)$ pour tout Δ tel que $0 < \Delta \leq maxdf\_end$. La plus grande valeur de cette fonction qui se trouve dans le dictionnaire $maxdf$ est égale à $maxdf\_end\_val = maxdf_M(m, \delta, maxdf\_end)$. Le dernier attribut, $next\_modes$, désigne l'ensemble de tous les états de module atteignables depuis le point d'hyperpériode situé à une distance $\delta_H - \delta$. Les états d'un module sont ainsi liés entre eux.

Attribut	Description	Type
$state\_end$	$\delta_H - \delta$, temps restant jusqu'à la prochaine hyperpériode δ_H	int
$df\_state\_end\_val$	$df_M(m, \delta, \delta + state\_end)$	int
$task\_activation$	vrai si une tâche est activée à (m, δ)	bool
$maxdf$	les demandes maximales, $maxdf[\Delta] : \Delta \rightarrow maxdf_M(m, \delta, \Delta)$	dict
$maxdf\_end$	max $arg(maxdf)$	int
$maxdf\_end\_val$	$maxdf[maxdf\_end]$	int
$next\_states$	les états de module atteignables depuis $(m, \delta + state\_end)$	ModuleState

TABLE 4.2 – Principaux attributs de la classe *ModuleState* pour l'état (m, δ) du module M

La structure du dictionnaire $maxdf$ doit davantage être précisée. La fonction de demande maximale est une fonction en escalier croissante. Elle ne croît qu'en certains points. Il est donc inutile de garder au sein du dictionnaire $maxdf$ des valeurs qui se répètent. Seules les valeurs correspondant à des points de discontinuité de la fonction de demande maximale

sont considérées. Le dictionnaire *maxdf* contient toutes les valeurs positives de la fonction de demande maximale pour Δ tel que [1] :

$$\begin{aligned} &1. \quad 0 < \Delta \leq maxdf\_end \\ &2. \quad 0 < maxdf_M(m, \delta, \Delta - 1) < maxdf_M(m, \delta, \Delta) \end{aligned} \tag{4.5}$$

Pour évaluer la fonction de demande maximale $maxdf_M(m, \delta, \Delta)$ selon l'Equation 4.3 dans tout intervalle Δ il est nécessaire de connaître les valeurs de la fonction de demande df_M entre le début de chaque état du module M et la première hyperpériode survenant après. Le dictionnaire *maxdf* doit alors contenir les valeurs de la fonction de demande au moins à cet instant. Dans ce cas, la portée du dictionnaire *maxdf* (*maxdf_end*) est égale ou supérieure à la prochaine hyperpériode (*state_end*) :

$$maxdf\_end \geq state\_end \tag{4.6}$$

4.5.2 Initialisation de l'espace de recherche

Une instance de la classe *ModuleState* doit être créée pour tout état d'un module correspondant à un démarrage d'une nouvelle tâche ou d'un nouveau mode. Les attributs de ces instances doivent être initialisés correctement. Il est précisé dans ce qui suit comment effectuer, pour chaque état (m, δ) d'un module M, la phase d'initialisation des variables d'une instance de la classe *ModuleStates* relative à cet état.

L'Algorithme 7 permet de remplir le dictionnaire *maxdf* de chaque instance de *ModuleState*, associée à un état de module, avec les valeurs de la fonction de demande jusqu'à la prochaine hyperpériode suivant immédiatement après cet état. L'algorithme, pour un mode m, calcule les valeurs de la fonction de demande $df_M(m, \delta, \delta')$ pour tous les temps de ce mode δ et δ' tels que :

- δ est inférieur à δ' ($\delta < \delta'$),
- δ est inférieur à la première hyperpériode dans le mode m ($0 \leq \delta < H[m]$),
- δ' est inférieur ou égal à la première hyperpériode dans le mode m ($0 < \delta' \leq H[m]$),
- δ coïncide avec lancement d'une tâche ou avec le début du mode m ($\delta = 0$),
- δ' coïncide avec l'échéance d'une tâche.

Ces résultats sont retournés sous la forme d'un dictionnaire *tableau_df*. Une entrée *tableau_df*$[\delta][\Delta]$ contient les valeurs de la fonction de demande $df_M(m, \delta, \delta + \Delta)$ où $\Delta > 0$ est la distance entre δ et un point de discontinuité de la fonction de demande :

$$df_M(m, \delta, \delta + \Delta - 1) < df_M(m, \delta, \delta + \Delta) \tag{4.7}$$

1. Rahni, Grolleau et Richard proposent de stocker les valeurs de la fonction de demande de la même façon pour le problème de faisabilité des transactions [141].

Lors de l'initialisation, ces résultats sont recopiés dans le dictionnaire *maxdf* de tout état dans le mode m.

Algorithme 7 *df_table*(m)

requiert: m est un mode dans le module M
assure: *tableau_df* est un tableau de valeurs de fonction de demande
$\forall(\delta,\delta') \in \{\{0 \cup m.task\_activations\} \times \{m.task\_deadlines\}\},\ \delta < \delta' \leq H[m]$,
$df_M(m,\delta,\delta') > df_M(m,\delta,\delta'-1) : \Delta = \delta' - \delta,\ tableau\_df[\delta][\Delta] = df_M(m,\delta,\delta')$

```
1: function DF_TABLE(m)
2:     tableau_df ← [][]
3:     pour tout δ ∈ {0 ∪ m.task_activations} : δ < H[m] faire
4:         last_df_val ← 0
5:         pour tout δ' ∈ m.task_deadlines : δ' ≤ H[m] faire
6:             si δ' > δ alors
7:                 df_val ← df_M(m, δ, δ')
8:                 si df_val > last_df_val alors
9:                     Δ ← δ' − δ
10:                    tableau_df[δ][Δ] ← last_df_val ← df_val
11:                fin si
12:            fin si
13:        fin pour tout
14:    fin pour tout
15:    renvoie tableau_df
16: fin function
```

Exemple 4.2. *Pour le mode m_1 dans le module M_1 du système de l'Exemple 3.15 l'Algorithme 7 retourne les valeurs suivantes : $tableau\_df[0][5] = 1$, $tableau\_df[0][6] = 3$, $tableau\_df[0][10] = 4$ et $tableau\_df[5][5] = 1$.*

L'Algorithme 8 fait appel à toutes les instances de la classe *ModuleState* pour chaque état du module M correspondant :

- au démarrage d'une nouvelle tâche,
- à l'activation d'un nouveau mode,
- à une fin d'hyperpériode (évaluation d'un commutateur de mode).

Toutes ces instances sont conservées dans le dictionnaire *mstates*. Pour chaque mode $m \in M.modes$, l'algorithme fait appel à la fonction *df_table*(m) décrite dans l'Algorithme 7. Le dictionnaire *tableau_df* qu'elle renvoie est ensuite utilisé pour remplir les dictionnaires des instances *ModuleState* de tous les états considérés dans ce mode. L'algorithme cherche pour cela tout temps δ du mode m, dans la limite de son hyperpériode $H[m]$, auquel une tâche est activée. Le début de ce mode ($\delta = 0$) est également considéré même si aucune tâche n'y démarre. Une entrée dans le dictionnaire *mstates* est créée pour tout δ. Cette entrée reçoit l'instance de la classe *ModuleStates* représentant l'état de module (m,δ). Les valeurs de $tableau\_df[\delta]$ sont recopiées dans son dictionnaire *maxdf*. Les variables *state_end*, *maxdf_end* et *task_activation* sont aussi mises à jour. Les instances de la classe *ModuleState* des états de module situés au-delà de la première hyperpériode

possèdent, mis à part leurs liens vers leurs états successeurs, les mêmes caractéristiques que les instances des états situés avant ce point. Les copies des instances obtenues précédemment sont alors insérées dans les entrées du dictionnaire *mstates* et décalées par un multiple de l'hyperpériode. La dernière étape consiste à lier les états entre eux. A chaque état de module (m, δ) sont associées les instances de *ModuleStates* des états de module atteignables depuis la fin de cet état. Les démarrages de nouveaux modes, dont la condition de commutateur est vérifiée à l'instant de la prochaine hyperpériode ($\delta_H = \delta + mstates[m][\delta].state\_end$), ainsi que la continuation dans le même mode à partir de cette hyperpériode sont pris en compte.

Algorithme 8 *Get_module_states*(M)

requiert: M est un module E-TDL
assure: *mstates* est le tableau de *ModuleStates* pour le module M
1: $mstates \leftarrow [\,][\,]$
2: **pour tout** $m \in M.modes$ **faire**
3: $df_vals \leftarrow$DF_TABLE(m)
4: **pour tout** $\delta \in \{\, 0 \cup m.task\_activations \,\} : \delta < H[m]$ **faire**
5: $mstates[m][\delta].maxdf \leftarrow df\_vals[\delta]$
6: $mstates[m][\delta].state\_end \leftarrow mstates[m][\delta].maxdf\_end \leftarrow H[m] - \delta$
7: $mstates[m][\delta].task\_activations \leftarrow vrai$
8: **fin pour tout**
9: **si** $0 \notin m.task\_activations$ **alors**
10: $mstates[m][0].task\_activation \leftarrow faux$
11: **fin si**
12: **si** $T[m] > H[m]$ **alors**
13: **pour tout** $n \leftarrow 2, T[m]/H[m]$ **faire**
14: **pour tout** $\delta \in \{\, 0 \cup m.task\_activations \,\} : \delta < H[m]$ **faire**
15: $mstates[m][\delta + (n-1)H[m]] \leftarrow mstates[m][\delta]$
16: **fin pour tout**
17: **fin pour tout**
18: **fin si**
19: **fin pour tout**
20: **pour tout** $m \in M.modes$ **faire**
21: **pour tout** $\delta \in \{\, \{\, (n-1)H[m] \mid n = 1 \ldots T[m]/H[m] \,\} \cup \{\, m.task\_activations \,\} \,\}$ **faire**
22: $fin \leftarrow \delta + mstates[m][\delta].state\_end$
23: $mstates[m][\delta].next\_states \leftarrow \{\, (m', 0) \mid m' \in prochains\_modes(m, fin) \,\}$
24: **si** $fin \neq T[m]$ **alors**
25: $mstates[m][\delta].next\_states \leftarrow mstates[m][\delta].next\_states \cup \{\, (m, fin + 1) \,\}$
26: **fin si**
27: **fin pour tout**
28: **fin pour tout**

Exemple 4.3. *L'Algorithme 8 peut être appliqué en reprenant le module M_1 du système de l'Exemple 3.15. Les instances de la classes ModuleState initialisées sont présentées sur la Figure 4.6.*

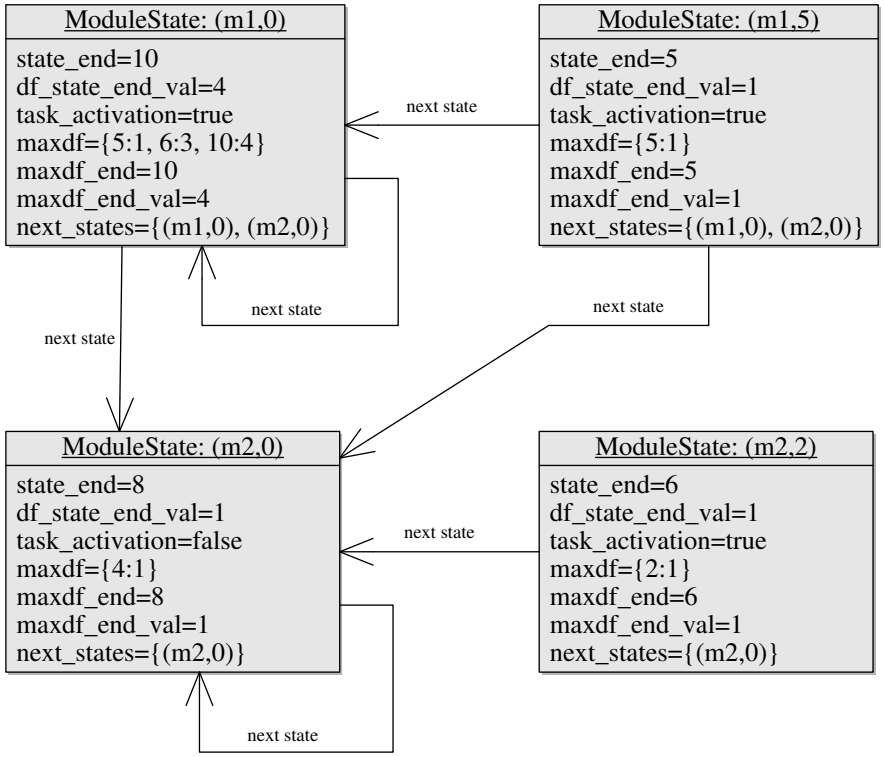


FIGURE 4.6 – Instances de la classe *ModuleState* pour le module M_1 du système de l'Exemple 3.15

4.5.3 Calcul de la demande maximale de processeur

Une méthode récursive de calcul de la demande maximale de processeur est proposée en section 4.5.3.1. Elle réalise le calcul de l'Equation 4.3. Afin de réduire sa complexité, une autre méthode, évitant les appels récursifs, est donnée en section 4.5.3.2. Les deux méthodes ne peuvent être appliquées qu'à des états de module dans lesquels une tâche ou un mode sont déclenchés. Un algorithme est proposé en section 4.5.3.3 afin de lever cette contrainte.

4.5.3.1 Une méthode récursive pour le calcul de demande maximale

On cherche la valeur de la fonction de demande maximale $maxdf_M(m, \delta, \Delta)$ pour l'état (m, δ) d'un module M dans un intervalle $\Delta > 0$. Dans ce qui suit, on suppose que cet état coïncide avec l'activation d'une tâche ou d'un mode. On suppose aussi que *mstates* est un dictionnaire des instances de *ModuleState* initialisées pour le module M par l'Algorithme 8. L'instance $mstates[m][\delta]$ correspond alors à l'état de module (m, δ) et est dénoté par ms. La fonction de demande maximale pour cet état dans l'intervalle Δ est calculée selon la méthode présentée sur la Figure 4.7 et, de manière plus détaillée, par l'Algorithme 9.
La méthode utilisée diffère selon la longueur de l'intervalle Δ :

$\Delta \leq$ ms.state_end :
La valeur de la fonction $maxdf_M(m, \delta, \Delta)$ peut être retrouvée dans le dictionnaire *ms.maxdf* (l'Algorithme 8 garantit que, pour toute instance initialisée, son dictionnaire *maxdf* contient toutes les valeurs de la fonction de demande jusqu'à *state_end*). Dans ce cas, il existe une seule et unique trace pouvant s'exécuter puisque aucun changement de mode ne peut se produire avant *ms.state_end*.

$\Delta >$ ms.state_end :
La demande maximale est la somme de :

- la demande produite jusqu'à la fin de *ms*
  (dans l'intervalle $[\,\delta, \delta + ms.state\_end\,]$),
- la demande maximale produite à partir de la fin de *ms*
  (dans l'intervalle $[\,\delta + ms.state\_end, \delta + ms.state\_end + \Delta'\,]$
  où $\Delta' = \Delta - state\_end$).

L'exécution qui se déroule jusqu'à la fin de *ms* suit toujours le même schéma et la demande qu'elle fait peut être déterminée par la valeur de *ms.df_state_end*. Par contre, à partir de la fin de *ms*, en fonction des états suivants (*ms.next_states*), il existe une ou plusieurs traces qui peuvent s'exécuter durant Δ'. C'est la demande maximale produite par une de ces traces qu'il convient de retenir. Afin de la trouver, on appelle, de manière récursive, la méthode proposée pour tous les états successifs de *ms* et pour l'intervalle Δ'.

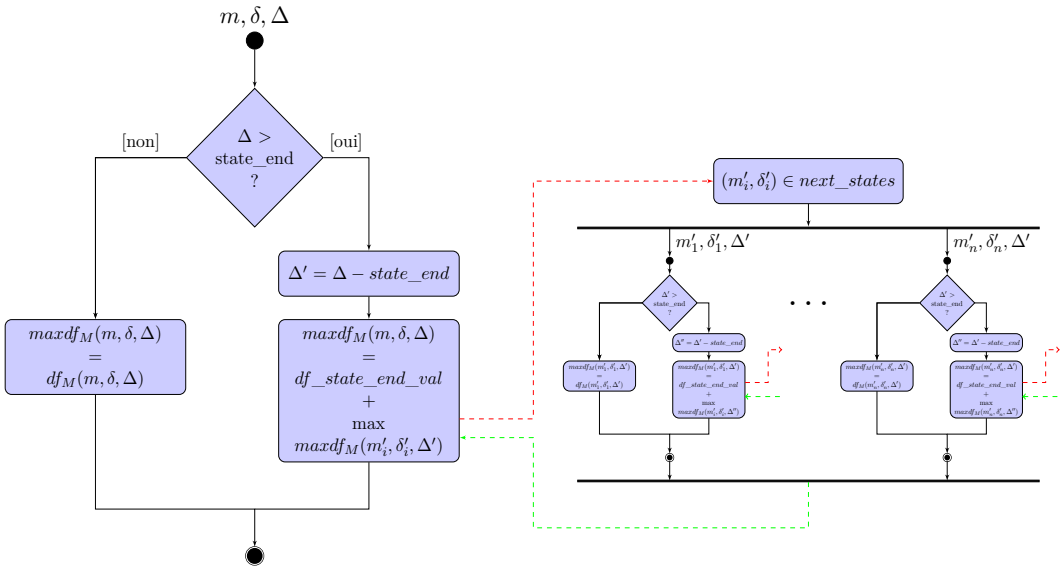


FIGURE 4.7 – Méthode récursive pour le calcul de la demande maximale (Algorithme 9)

Algorithme 9 $maxdf(m, \delta, \Delta, mstates)$

requiert: m est un mode dans un module E-TDL $M : m \in M.modes$
requiert: δ est le temps de mode $m : 0 \leq \delta \leq T[m]$ et $\Delta > 0$
requiert: $mstates$ est le dictionnaire contenant les instances de *ModuleState* pour le module M (retourné par l'Algorithme 8)
requiert: une instance de *ModuleState* pour l'état (m, δ) existe dans $mstates : mstates[m][\delta] \neq \emptyset$
assure: la valeur retournée est égale à $maxdf_M(m, \delta, \Delta)$

function MAXDF$(m, \delta, \Delta, mstates)$
  $ms \leftarrow mstates[m][\delta]$
  $maxdf\_val \leftarrow 0$
  si $\Delta \leq ms.state\_end$ **alors**
    pour tout $\Delta_i \in \arg(ms.maxdf)$ **faire**
      si $\Delta_i \leq \Delta$ **alors**
        $maxdf\_val \leftarrow ms.maxdf[\Delta_i]$
      sinon
        break
      fin si
    fin pour tout
  sinon
    $\Delta' \leftarrow \Delta - ms.state\_end$
    $maxdf\_val \leftarrow ms.df\_state\_end\_val +$
      $+ \max \{ maxdf(m', \delta', \Delta', mstates) \mid (m', \delta') \in ms.next\_states \}$
  fin si
  renvoie $maxdf\_val$
fin function

Exemple 4.4. *Soient les instances de ModuleState initialisées dans l'Exemple 4.3 pour le module M_1 du système de l'Exemple 3.15. On cherche la valeur de la fonction de demande maximale de l'état $(m_1, 5)$ du module sur un intervalle de 19 : $maxdf_{M_1}(m_1, 5, 19)$. Le déroulement de la procédure récursive pour le calcul de la demande maximale est illustré par la Figure 4.8. Les arcs illustrent les appels à la procédure de calcul pour les états suivants (next_states). Ces appels sont réalisés pour des valeurs d'intervalle adéquatement réduites (voir sur les côtés de la Figure). La séquence d'exécution qui donne la plus grande demande de temps processeur est en rouge.*

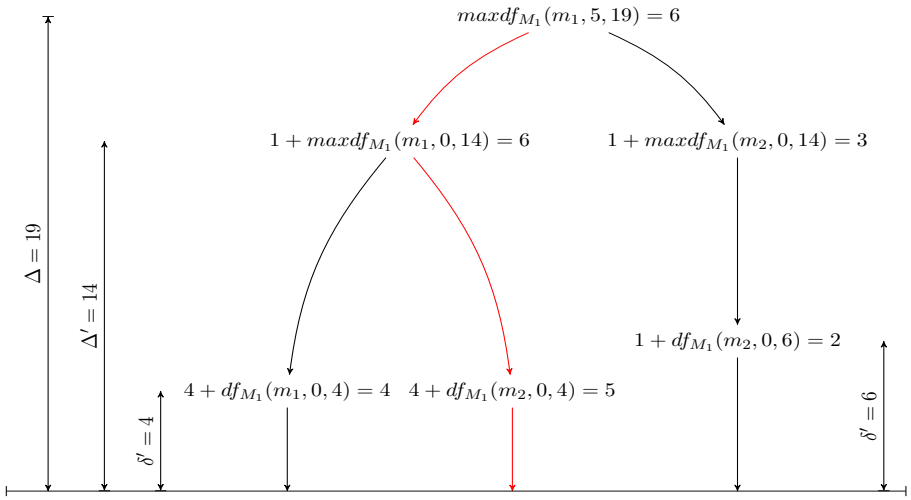


FIGURE 4.8 – Calcul de la fonction de demande maximale $maxdf_{M_1}(m_1, 5, 19)$ selon l'Algorithme 9

4.5.3.2 Une méthode non-récursive pour le calcul de demande maximale

Lors de l'exécution du test d'ordonnançabilité, tous les intervalles dont les longueurs sont comprises entre 0 et la borne de faisabilité $\Delta_{max}(M)$ sont à vérifier. En supposant que la demande maximale est évaluée dans l'ordre croissant des intervalles, si à un certain point de l'analyse, les valeurs de la fonction de demande maximale $maxdf_M(m, \delta, \Delta)$ sont évaluées pour tous les états du module M dans l'intervalle Δ, alors les valeurs de la fonction de demande maximale $maxdf_M(m, \delta, \Delta')$ sont déjà connues pour tous les états du module M dans tous les intervalles Δ' inférieurs à Δ. Au lieu de procéder à un appel récursif à chaque fois que l'on a besoin de valeurs de demande maximale pour des intervalles inférieurs à celui dont on cherche la demande maximale, ces valeurs peuvent être récupérées à partir des résultats calculés lors d'appels précédents. Les valeurs de la fonction de demande maximale sont dès lors conservées pour tout intervalle. En pratique chaque instance de *ModuleState*

pour laquelle la méthode est appelée dans un intervalle Δ plus grand que les intervalles évalués antérieurement pour cette instance, est modifiée. Cette modification est réalisée de la façon suivante :

- la valeur de demande maximale obtenue pour Δ est insérée dans le dictionnaire *maxdf*
- les variables *maxdf_end* et *maxdf_end_val* sont mises à jour

Ainsi, le nombre d'appels récursifs est limité à un seul appel pour chaque état successeur (*next_states*). La Figure 4.9 et l'Algorithme 10 exposent précisément la mise en œuvre de cette méthode.

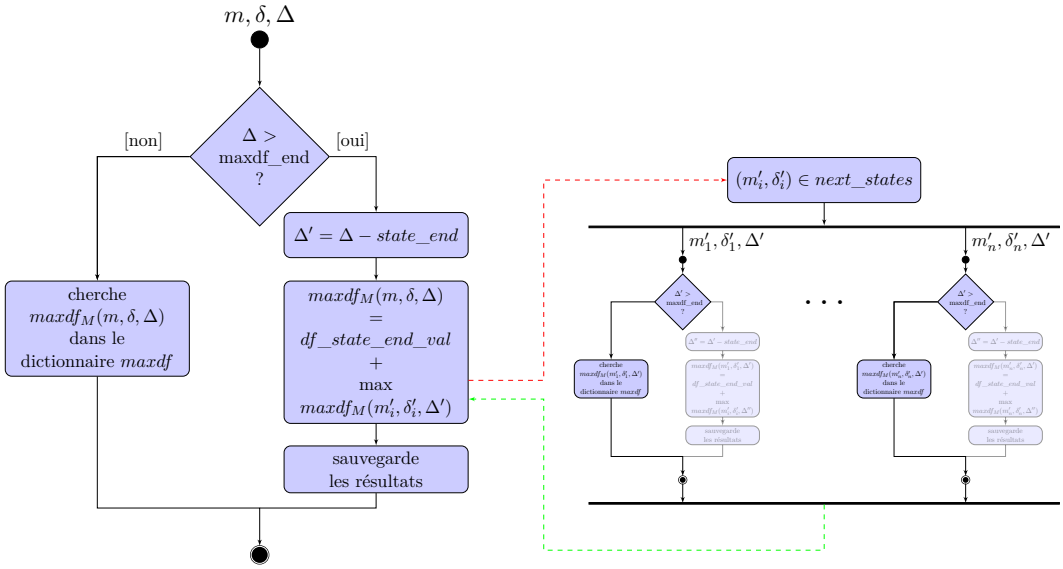


FIGURE 4.9 – Méthode non-récursive pour le calcul de demande maximale (Algorithme 10)

Exemple 4.5. *On examine de nouveau le calcul de la fonction de demande maximale* $maxdf_{M_1}(m_1, 5, 19)$ *dans l'Exemple 4.4. Ce calcul est réalisé en utilisant la méthode proposée dans l'Algorithme 10. Le fonctionnement de la procédure est illustré sur la Figure 4.10. On suppose que le calcul de demande maximale a été déjà réalisé pour tous les intervalles inférieurs à 19 et que les résultats de ce calcul sont stockés dans les dictionnaires des instances de ModuleState de chaque état du module. Lorsque la méthode s'appelle à elle-même pour évaluer les demandes dans l'intervalle de longueur* 14 *(*$maxdf_{M_1}(m_1, 0, 14)$ *et* $maxdf_{M_1}(m_2, 0, 14)$*), les valeurs de ces demandes sont disponibles dans les dictionnaires maxdf (*$mstates[m_1][0].maxdf$ *et* $mstates[m_2][0].maxdf$*). Une fois le résultat de* $maxdf_{M_1}(m_1, 5, 19)$ *obtenu, le dictionnaire maxdf associé à l'état de module* $(m_1, 5)$ *est mis à jour avec ce résultat.*

Algorithme 10 $maxdf(m, \delta, \Delta, mstates)$

requiert: m est un mode dans un module E-TDL $M : m \in M.modes$
requiert: δ est le temps de mode $m : 0 \leq \delta \leq T[m]$ et $\Delta > 0$
requiert: $mstates$ est le dictionnaire contenant les instances de *ModuleState* pour le module M (retourné par l'Algorithme 8) et $\forall ms \in mstates : ms.maxdf\_end \geq \Delta - 1$
requiert: une instance de *ModuleState* pour l'état (m, δ) existe dans $mstates : mstates[m][\delta] \neq \emptyset$
assure: la valeur retournée est égale à $maxdf_M(m, \delta, \Delta)$
assure: $mstates[m][\delta].maxdf\_end = \Delta$
assure: si $maxdf_M(m, \delta, \Delta) > mstates[m][\delta].maxdf\_end\_val$ alors
$mstates[m][\delta].maxdf\_end\_val = mstates[m][\delta].maxdf[\Delta] = maxdf_M(m, \delta, \Delta)$

```
1: function MAXDF(m, δ, Δ, mstates)
2:     ms ← mstates[m][δ]
3:     maxdf_val ← 0
4:     si Δ ≤ ms.maxdf_end alors
5:         pour tout Δ_i ∈ arg(ms.maxdf) faire
6:             si Δ_i ≤ Δ alors
7:                 maxdf_val ← ms.maxdf[Δ_i]
8:             sinon
9:                 break
10:            fin si
11:        fin pour tout
12:    sinon
13:        Δ' ← Δ − ms.state_end
14:        maxdf_val ← ms.df_state_end_val +
                       + max{ maxdf(m', δ', Δ', mstates) | (m', δ') ∈ ms.next_states }
15:        si maxdf_val > ms.maxdf_end_val alors
16:            ms.maxdf[Δ] ← ms.maxdf_end_val ← maxdf_val
17:        fin si
18:        ms.maxdf_end ← Δ
19:    fin si
20:    renvoie maxdf_val
21: fin function
```

Pour évaluer et comparer les performances des différents algorithmes de calcul de demande maximale proposés dans cette section, le module M_1 du système décrit à l'Exemple 3.15 est considéré. Pour chacun des algorithmes, le calcul de demande maximale pour toutes les instances de *ModuleState* initialisées par l'Algorithme 8 est réalisé. La Figure 4.11 indique le temps nécessaire pour exécuter le calcul de demande maximale dans tous les intervalles inférieurs ou égaux à Δ.
Cette expérimentation a été réalisée sur une machine fonctionnant sous Fedora release 17 (Beefy Miracle) équipée de 4 processeurs Intel Xeon W3530 @ 2.80GHz.

4.5.3.3 Calcul de demande maximale pour un état de module arbitraire

Les algorithmes proposés ci-dessus fournissent les valeurs de la fonction de demande maximale seulement pour les états d'un module correspondant à une certaine activité de ce module (voir Algorithme 8). L'approche proposée ci-après permet d'évaluer la fonction de demande maximale pour tout état de module, même si cet état n'est lié à aucune activité

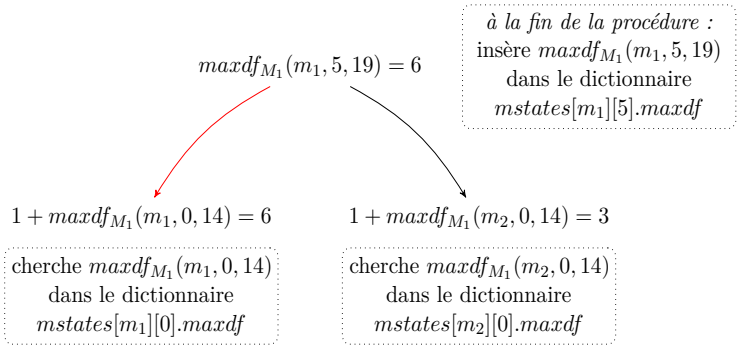


FIGURE 4.10 – Calcul de la fonction de la demande maximale $maxdf_{M_1}(m_1, 5, 19)$ selon l'Algorithme 10

du module. Cette approche est détaillée dans l'Algorithme 11.

L'algorithme prend en entrée l'état (m, δ) d'un module M, un intervalle Δ et un tableau *mstates* comportant les instances de *ModuleState* initialisées par l'Algorithme 8. Si l'état (m, δ) est décrit dans *mstates*, la valeur de la fonction de demande maximale est trouvée, comme dans le cas précédent, en faisant appel à la fonction $maxdf(m, \delta, \Delta, mstates)$ (Algorithme 10). Dans le cas contraire, l'algorithme cherche les états situés immédiatement après l'état (m, δ) et dont les instances de *ModuleState* existent dans le tableau *mstates*. L'algorithme détermine d'abord trois temps de mode significatifs :

δ_p le temps de mode qui précède immédiatement δ et une instance de *ModuleState* est créée dans *mstates* pour (m, δ_p)

δ_n le temps de mode qui succède immédiatement à δ et une instance de *ModuleState* est créée dans *mstates* pour (m, δ_n)

δ_{sw} le premier instant de changement de mode après δ

La connaissance des états de module qui peuvent suivre immédiatement l'état (m, δ) et pour lesquels les instances de *ModuleState* sont créées dans le tableau *mstates* permet d'évaluer la demande maximale $maxdf_M(m, \delta, \Delta)$. Il convient alors de considérer deux cas :

$\delta_n = \delta_{sw}$ Un changement de mode peut se produire immédiatement et les états dans lesquels l'exécution peut se poursuivre sont donnés par $mstates[m][\delta_p].next\_states$. Chacun de ces états est atteignable après un intervalle $\delta_{sw} - \delta$. La demande maximale $maxdf_M(m, \delta, \Delta)$ est égale à la plus grande des demandes maximales de ces états dans l'intervalle $\Delta' = \Delta - (\delta_{sw} - \delta)$.

$\delta_n < \delta_{sw}$ L'exécution passe d'abord par l'état (m, δ_n) avant l'évaluation d'un commutateur de mode. Cet état est atteignable après l'intervalle $\delta_n - \delta$. La

demande maximale $maxdf_M(m, \delta, \Delta)$ est égale à la demande maximale $maxdf_M(m, \delta_n, \Delta')$ où $\Delta' = \Delta - (\delta_n - \delta)$.

Algorithme 11 $maxdf\_any(m, \delta, \Delta, mstates)$

requiert: m est un mode dans un module E-TDL $M : m \in M.modes$
requiert: δ est le temps de mode $m : 0 \leq \delta \leq T[m]$ et $\Delta > 0$
requiert: $mstates$ est le dictionnaire comportant les instances de *ModuleState* pour le module M (retourné par l'Algorithme 8)
assure: la valeur retournée est égale à $maxdf_M(m, \delta, \Delta)$

function MAXDF_ANY($m, \delta, \Delta, mstates$)
  si $\delta \in \{ \{ (k-1)H[m] \mid k = 1 \dots T[m]/H[m] \} \cup \{ m.task\_activations \} \}$ **alors**
    renvoie $maxdf(m, \delta, \Delta)$
  fin si
  pour tout $\delta_i \in \{ \{ (k-1)H[m] \mid k = 1 \dots T[m]/H[m] \} \cup \{ m.task\_activations \} \}$ **faire**
    si $\delta_i > \delta$ **alors**
      $\delta_n \leftarrow \delta_i$
      break
    fin si
    $\delta_p \leftarrow \delta_i$
  fin pour tout
  $\delta_{sw} \leftarrow \delta_p + mstates[m][\delta_p].state\_end$
  si $\delta_n = \emptyset$ ou $\delta_n = \delta_{sw}$ **alors**
    $\Delta' \leftarrow \Delta - (\delta_{sw} - \delta)$
    renvoie $\max \{ maxdf(m', \delta', \Delta') \mid (m', \delta') \in mstates[m][\delta_p].next\_states \}$
  sinon
    $\Delta' \leftarrow \Delta - (\delta_n - \delta)$
    renvoie $maxdf(m, \delta_n, \Delta')$
  fin si
fin function

Exemple 4.6. *Les valeurs de la fonction de demande maximale pour les deux états du module M_1 du système introduit dans l'Exemple 3.15 sont calculées. Les instances de ModuleState pour ces états ne sont pas initialisées par l'Algorithme 8.*

- $maxdf_{M_1}(m_1, 1, 23) = maxdf_{M_1}(m_1, 5, 19) = 6$
- $maxdf_{M_1}(m_1, 9, 10) = \max \{ maxdf_{M_1}(m_1, 0, 9), maxdf_{M_1}(m_2, 0, 9) \} = \max\{ 3, 1 \} = 3$

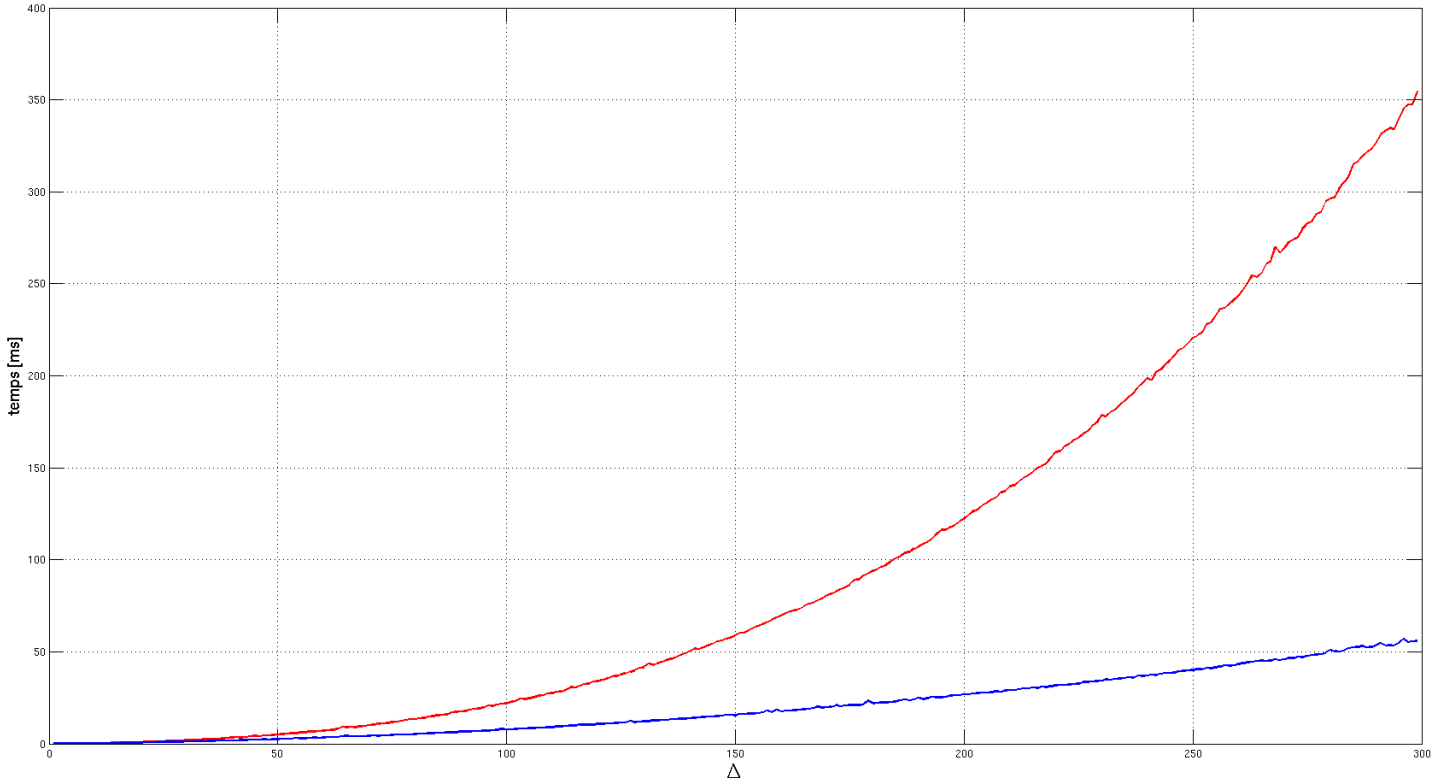


FIGURE 4.11 – Temps du calcul de $maxdf_{M_1}(m, \delta, \Delta)$ pour le module M_1 du système de l'Exemple 3.15 : en rouge selon l'Algorithme 9, en bleu selon l'Algorithme 10

4.6 Test d'ordonnançabilité

Cette section décrit le fonctionnement du test d'ordonnançabilité pour un ensemble de modules E-TDL sous *EDF* dans un contexte monocœur. Le test suit les étapes de la Figure 4.1. Trois étapes sont essentiellement identifiées : initialisation, analyse approchée et analyse exacte. Ces trois étapes sont détaillées respectivement dans les sections 4.6.2, 4.6.3 et 4.6.4.

Algorithme 12 Test d'ordonnançabilité pour un ensemble de modules E-TDL sous EDF

requiert: *Modules* est l'ensemble des modules E-TDL
1: **Initialisation**
2: **pour tout** $\Delta \in 1 \dots \Delta_{max}$ **faire**
3: ordonnançable $\leftarrow$ **Analyse approchée**
4: **si** non ordonnançable **alors**
5: ordonnançable $\leftarrow$ **Analyse exacte**
6: **si** non ordonnançable **alors**
7: **renvoie** Faux
8: **fin si**
9: **fin si**
10: **fin pour tout**
11: **renvoie** Vrai

4.6.1 Cas particuliers

Le problème de l'ordonnançabilité peut être résolu, selon le modèle du système et selon le modèle des taches, en appliquant les différents tests dont la complexité est moins importante que celle du test proposé :

- $|\mathbf{Modules}| = \mathbf{1}$ si le système exécute un seul module, le Théorème 1 permet de vérifier l'ordonnançabilité du système en analysant l'ordonnançabilité de chacun de ses modes séparément ;
- $\mathbf{TEL} = \mathbf{T}$ si les tâches du système ont leur période égale à leur Temps d'Exécution Logique (*TDL* classique), une condition nécessaire et suffisante d'ordonnançabilité est donnée par l'Equation 2.53 (fondée sur le taux d'utilisation).

Les tests adaptés au traitement des deux cas mentionnés ci-dessus sont fournis dans l'outil. L'outil détecte au préalable quel est le type du système et applique le test lui correspondant. Ce qui suit porte sur des systèmes nécessitant un test complet.

Même si toutes les tâches ont leurs offsets égaux à 0 (mais $TEL \leq T$) l'instant critique ne correspond pas forcement aux débuts des modes. L'exemple suivant illustre un tel cas.

Exemple 4.7. *Soit un module M composé de deux modes m_1 et m_2 dont les périodes sont égales à 6. Le mode m_1 exécute deux tâches : $\tau_{11} = (0, 1, 3, 3)$ $\tau_{12} = (0, 1, 3, 6)$. Le mode m_2*

exécute une seule tâche : $\tau_{21} = (0, 3, 3, 6)$. *Les offsets de toutes les tâches dans ce module sont alors égaux à* 0.

On cherche la plus grande demande de temps processeur qui peut être produite par les tâches de ce module dans tout intervalle de durée $\Delta = 6$. *Pour les intervalles de temps dont les débuts coïncident avec un début de mode (toutes les tâches de ce mode y commencent alors leur exécution puisque leurs offsets sont égaux à* 0*) les fonctions de demande sont suivantes :* $df_M(m_1, 0, 6) = 3$ *et* $df_M(m_2, 0, 6) = 3$. *Pour l'intervalle qui commence à l'instant* 3 *du mode* m_1 *et se termine à l'instant* 3 *du mode* m_2 *(le module* M *change de mode dans cet intervalle), la fonction de demande est égale à :* $maxdf_M(m_1, 3, 6) = df_M(m_1, 3, 6) + df_M(m_2, 0, 3) = 1 + 3 = 4$. *Alors, pour* $\Delta = 6$, *la demande dans l'intervalle qui commence à l'instant* 3 *du mode* m_2 *dépasse les demandes dans les intervalles qui commencent aux instants de débuts des modes* m_1 *et* m_2.

4.6.2 Initialisation

La mise en œuvre du test implique l'utilisation des structures de données suivantes (Δ est la longueur de l'intervalle traité) :

- **mstates$_M$** le tableau de $ModuleStates$ pour le module M
- **mdbf_val$_M$** la valeur de $mbdf_M(\Delta)$
- **mdbf_modes$_M$** l'ensemble de modes dans lesquels le module M peut produire la plus grande demande maximale dans un intervalle de durée Δ
- **sum_mbdf** la somme de $mdbf_M(\Delta)$ pour tous les modules $Modules$
- **p_confs** l'ensemble des configurations parallèles correspondant à l'activation d'une tâche
- **sum_maxdf$_\zeta$** la somme des demandes maximales sum_maxdf_ζ produites dans tous les modules de l'ensemble $Modules$ à partir de la configuration parallèle ζ
- **offsets** la liste de tous les décalages entre démarrages de modes des $Modules$
- **syncs** un ensemble de modes non-corrélés pour l'ensemble $Modules$

Lors de la première phase du test, ces variables ne contiennent aucune valeur. Le test commence par vérifier le taux d'utilisation (Equations 3.26, 3.52, 3.58). Si cette condition est satisfaite, il détermine la borne de faisabilité Δ_{max} et, grâce à l'Algorithme 8, initialise les tableaux $mstates_M$. Les autres variables sont traitées au cours des étapes suivantes du test.

4.6.3 Analyse approchée

L'analyse approchée d'ordonnançabilité d'un ensemble $Modules$ de modules E-TDL dans l'intervalle Δ consiste à :

1. évaluer la somme des bornes de fonctions de demandes maximales sum_mbdf pour tous les modules $M \in Modules$ dans cet intervalle :

$$sum\_mbdf \leftarrow \sum_{M \in Modules} mdbf_M(\Delta)$$

2. vérifier que cette somme ne dépasse pas le temps de traitement $sbf(\Delta)$ disponible dans l'intervalle Δ :

$$sum\_mbdf \leq sbf(\Delta)$$

Si cette condition n'est pas vérifiée, l'exécution du test est redirigée vers l'analyse exacte.

Cette partie du test est décrite par l'Algorithme 13. Celui-ci fait appel à la fonction *compute_mdbf*. La fonction *compute_mdbf* permet de déterminer pour chaque module M la valeur de $mdbf_M(\Delta)$. Elle est spécifiée en détail dans l'Algorithme 14.

Algorithme 13 Analyse approchée

requiert: $Modules$ est un ensemble de modules E-TDL
requiert: Δ est un intervalle de temps, $\Delta > 0$
requiert: $\forall M \in Modules$: $mstates_M$ est le tableau de $ModuleStates$ pour le module M
1: $sum\_mdbf \leftarrow 0$
2: **pour tout** $M \in Modules$ **faire**
3: $\quad mdbf\_val_M, mdbf\_modes_M \leftarrow$ COMPUTE_MDBF$(M, \Delta, mstates_M)$
4: $\quad sum\_mdbf \leftarrow sum\_mdbf + mdbf\_val_M$
5: **fin pour tout**
6: **si** $sum\_mdbf > sbf(\Delta)$ **alors**
7: $\quad$ Analyse exacte
8: **sinon**
9: $\quad$ Ordonnançable pour Δ
10: **fin si**

La fonction $compute\_mdbf(M, \Delta, mstates_M)$, outre la valeur de demande maximale $mdbf\_val_M = mdbf_M(\Delta)$ émise dans le module M, retourne également un ensemble $mdbf\_modes_M$ de modes de ce module. Dans tout mode de cet ensemble commence un intervalle de durée Δ pour lequel la demande maximale est $mdbf\_val_M$. Cette information est utilisée lors de l'analyse exacte décrite dans la section suivante.

La fonction calcule d'abord, pour chaque état du module (m, δ) coïncidant avec le déclenchement d'une tâche, la fonction de demande maximale $maxdf_M(m, \delta, \Delta)$. Ce calcul est réalisé avec l'Algorithme 10. Si la valeur de $maxdf_M(m, \delta, \Delta)$ est plus grande que les valeurs des demandes maximales évaluées pour les états précédents du module, la variable $mdbf\_val_M$ et l'ensemble $mdbf\_modes_M$ sont mis à jour. Si cette valeur est égale à la plus grande valeur de demande maximale évaluée pour les états précédents du module, le mode m est ajouté à l'ensemble $mdbf\_modes_M$.

Exemple 4.8. *Pour le module M_1 du système de l'Exemple 3.15 (voir aussi la Figure 4.6 dans l'Exemple 4.3) l'Algorithme 14 produit, pour $\Delta = 5$, les résultats suivants :* $mdbf\_val_{M_1} = 1$ *et* $mdbf\_modes_{M_1} = \{ m_1, m_2 \}$.

Algorithme 14 *compute_mdbf*$(M, \Delta, mstates_M)$

requiert: M est un module E-TDL
requiert: Δ est un intervalle de temps, $\Delta > 0$
requiert: $mstates_M$ est le tableau de $ModuleStates$ des états du module M
assure: $mdbf\_val_M = mdbf_M(\Delta)$ et $\forall m \in mdbf\_modes_M \; \exists \; \delta : \; maxdf_M(m, \delta, \Delta) = mdbf_M(\Delta)$

```
 1: function COMPUTE_MDBF(M, Δ, mstates_M)
 2:     mdbf_val_M    ← 0
 3:     mdbf_modes_M ← ∅
 4:     pour tout m ∈ M.modes faire
 5:         pour tout δ ∈ { m.task_activations } faire
 6:             maxdf_val ← MAXDF(m, δ, Δ, mstates_M)
 7:             si maxdf_val > mdbf_val_M alors
 8:                 mdbf_val_M    ← maxdf_val
 9:                 mdbf_modes_M ← { m }
10:             fin si
11:             si maxdf_val = mdbf_val_M alors
12:                 mdbf_modes_M ← mdbf_modes_M ∪ { m }
13:             fin si
14:         fin pour tout
15:     fin pour tout
16:     renvoie mdbf_val_M, mdbf_modes_M
17: fin function
```

4.6.4 Analyse exacte

L'analyse exacte de l'ordonnançabilité d'un ensemble $Modules$ de modules *E-TDL* dans un intervalle Δ comprend :

1. le calcul d'un ensemble p_confs de configurations parallèles telles que chacune d'elles corresponde à l'activation d'une tâche,
2. les évaluations des sommes des fonctions de demandes maximales $maxdf_M(m, \delta, \Delta)$ pour les états de module appartenant à la même configuration parallèle $\zeta \in p\_confs$:

$$\forall \zeta \in p\_confs \; : \; sum\_maxdf_\zeta \leftarrow \sum_{(m,\delta)\in\zeta} maxdf_M(m, \delta, \Delta)$$

3. la vérification, pour toute configuration parallèle $\zeta \in p\_confs$, que la somme sum_maxdf_ζ obtenue pour celle-ci ne dépasse pas le temps de traitement $sbf(\Delta)$ disponible dans l'intervalle Δ :

$$\forall \zeta \in p\_confs \; : \; sum\_maxdf_\zeta \; \leq \; sbf(\Delta)$$

   S'il existe une configuration parallèle ζ pour laquelle cette condition n'est pas remplie, le système est infaisable.

Les points ci-dessus sont réalisés par l'Algorithme 15. Il retourne une réponse exacte à la question de l'ordonnançabilité d'un ensemble de modules E-TDL dans un intervalle Δ.

L'algorithme commence par initialiser les dictionnaires $offsets$ et $syncs$ s'ils sont vides. A cet effet, la fonction *Get_offsets* est appelée (voir Algorithme 5). A partir de

Algorithme 15 Analyse exacte

requiert: $Modules$ est un ensemble de modules E-TDL
requiert: Δ est un intervalle de temps, $\Delta > 0$
requiert: $\forall M \in Modules$: $mstates_M$ est le tableau de $ModuleStates$ associé au module M
1: **si** $offsets = \emptyset$ **alors**
2: $offsets, syncs \leftarrow$ GET_OFFSETS($Modules$)
3: **fin si**
4: **si** $\prod_{M \in Modules} mdbf\_modes_M \cap sync \neq \emptyset$ **alors**
5: **renvoie** non ordonnançable
6: **fin si**
7: **si** $p\_confs = \emptyset$ **alors**
8: $p\_confs \leftarrow$ GET_PARALLEL_CONFIGURATIONS($Modules, offsets$)
9: **fin si**
10: **pour tout** $\zeta \in p\_confs$ **faire**
11: $sum\_maxdf_\zeta \leftarrow 0$
12: **pour tout** $(m_i, \delta_i) \in \zeta$ **faire**
13: $sum\_maxdf_\zeta \leftarrow sum\_maxdf_\zeta +$ MAXDF_ANY($m_i, \delta_i, \Delta, mstates_{M_i}$)
14: **fin pour tout**
15: **si** $sum\_maxdf_\zeta > sbf(\Delta)$ **alors**
16: **renvoie** Non ordonnançable
17: **fin si**
18: **fin pour tout**
19: **renvoie** Ordonnançable pour Δ

20: **function** GET_PARALLEL_CONFIGURATIONS($Modules, offsets$)
21: $p\_confs \leftarrow \emptyset$
22: **pour tout** $M \in Modules$ **faire**
23: **pour tout** $m \in M.modes$ **faire**
24: **pour tout** $\delta \in m.task\_activations$ **faire**
25: $p\_confs \leftarrow p\_confs \cup$ SHIFT((m, δ),offsets)
26: **fin pour tout**
27: **fin pour tout**
28: **fin pour tout**
29: **renvoie** p_confs
30: **fin function**

syncs, il peut être déterminé s'il existe une configuration de modes parallèles pendant lesquels les modules génèrent simultanément leurs demandes maximales $mdbf_M(\Delta)$ (ligne 4). Si tel est le cas, le résultat de l'analyse approchée est en fait exact et l'ensemble des modules est infaisable. Sinon, il est toujours utile de procéder à la suite du test en vue d'obtenir des valeurs cumulatives de demandes maximales plus précises (en espérant que celles-ci sont en réalité inférieures à la valeur sum_mdbf). A cet effet, il est nécessaire de trouver toutes les configurations parallèles p_confs coïncidant avec l'activation d'une tâche (la fonction *get_parallel_configurations* fait appel à la fonction *shift* décrite dans l'Algorithme 6). Ensuite, pour chacune de ces configurations $\zeta \in p\_confs$, la seconde condition (Equation 3.53) du Théorème 3 (ou de sa version pour les serveurs, l'Equation 3.59 du Théorème 4) est vérifiée. Pour ce faire, la somme des demandes maximales sum_maxdf_ζ, produites dans les différents modules à partir de cette configuration durant l'intervalle Δ,

est calculée. Cela nécessite de faire appel à la fonction *maxdf_any* (voir Algorithme 11). Cette fonction opère sur les dictionnaires $mstates_{M_i}$ traités auparavant lors de l'analyse approchée pour la même durée d'intervalle Δ. Par conséquent, la fonction dispose des valeurs des demandes maximales dans cet intervalle pour tous les états de mode significatifs (i.e. qui correspondent à une activation d'une tâche ou d'un mode). Cela permet d'effectuer cette opération de manière efficace. Si pour l'une des configurations parallèles ζ la somme sum_maxdf_ζ est supérieure au temps de processeur $sbf(\Delta)$ disponible dans l'intervalle Δ, le système est infaisable. Dans le cas contraire (la vérification n'échoue pour aucune des configurations parallèles p_confs), le système est prouvé ordonnançable dans tous les intervalles de durée Δ.

Exemple 4.9. *Soit le système de l'Exemple 3.15. L'outil est utilisé pour vérifier son ordonnançabilité. Le test a été exécuté sur une machine fonctionnant sous Fedora release 17 (Beefy Miracle) équipée de 4 processeurs Intel Xeon W3530 @ 2.80GHz. Les résultats sont obtenus au but de 0.428s. La borne de faisabilité est égale à 54. L'analyse exacte est réalisée pour les intervalles de durée 2 et 3. Pour les intervalles de durée autre, l'analyse approchée s'avère suffisante.*

4.7 Validation par simulation

Cette section présente un simulateur pour les systèmes modélisés en *E-TDL*. Le simulateur permet de tracer toutes les séquences d'exécution dans ces systèmes. Il reproduit les décisions d'un ordonnanceur en tenant compte de tous les scénarios possibles d'évolution du système (changements de modes).

4.7.1 Simulateurs disponibles

Il existe déjà un certain nombre de simulateurs d'ordonnancement.

Storm [173] est un simulateur d'ordonnancement multiprocesseur développé en Java. L'utilisateur a la possibilité d'ajouter facilement de nouvelles politiques d'ordonnancement ainsi que d'étendre le modèle de tâche temps-réel. Pour chaque ensemble de tâches il produit des traces d'exécution sous forme de diagrammes de Gantt.

Cheddar [157] est un outil conçu en Ada qui rassemble des tests d'ordonnançabilité et un environnement de simulation pour vérifier le respect de contraintes temporelles d'un système. Le système est défini en spécifiant le nombre de ses processeurs, leurs caractéristiques, les files d'attente, les tâches (avec leurs précédences) et les ressources partagées.

SimSo [47, 48, 49, 50] est un outil de simulation pour des architectures multiprocesseurs mettant à la disposition de l'utilisateur plus de vingt-cinq algorithmes d'ordonnancement. *SimSo* est développé en Python sur la base de la bibliothèque *SimPy*.

L'outil prend en compte les effets du cache ainsi que le coût de la préemption sur le temps d'exécution des tâches. Il peut être facilement étendu afin d'intégrer de nouvelles politiques d'ordonnancement et de nouveaux modèles des tâches. De plus, il permet d'automatiser la génération d'ensembles de tâches et le déroulement de l'évaluation des politiques d'ordonnancement.

RTSIM [128, 129] est un jeu de bibliothèques écrites en C++ visant la simulation commune d'un système dynamique et d'une plateforme exécutant une tâche du contrôleur de ce système. Ceci permet d'étudier l'impact de l'utilisation de plateformes d'exécution dotées de caractéristiques différentes (vitesse de traitement, politique d'ordonnancement etc.) sur la performance du processus de contrôle.

YARTISS [46] est un simulateur pour des architectures multiprocesseurs développé en Java. Yartiss permet d'évaluer et de comparer des politiques d'ordonnancement du point de vue de leur consommation énergétique. Il est également possible de définir des tâches en s'appuyant sur un modèle de graphe acyclique.

Tous ces simulateurs offrent un éventail de fonctionnalités permettant d'analyser différents aspects des systèmes temps-réel. Cela dit, il semble difficile d'intégrer, au sein de ces simulateurs, un mécanisme de changement de mode tel qu'utilisé dans les langages de définition temporelle. Une autre possibilité, qui se prêterait mieux à la mise en œuvre de ce mécanisme, serait de s'orienter vers un outil de *model checking* comme, par exemple, *Uppaal* [28]. La structure des automates temporisés [7], disponible en *Uppaal*, permet d'exprimer des activités à déclenchement temporel (par exemple, *Times* [9, 10] est un outil pour la modélisation et la vérification de systèmes temps-réel qui repose sur *Uppaal*). Cependant, il apparaît plus aisé, comme solution, d'utiliser l'interface d'E-TDL déjà existante (voir section 4.2) pour donner une base à un nouveau simulateur.

4.7.2 Caractérisation de l'état d'un système

Un système exécutant un ensemble de modules E-TDL peut être, à tout moment, caractérisé par sa configuration parallèle (section 3.6). C'est une vision logique de son exécution. Elle ne prend pas en compte son aspect physique. Cet aspect est déterminé par des instances de tâches qui sont actives à un instant donné et par l'état de leur exécution observé à cet instant. Les paragraphes qui suivent permettent de caractériser un état du système et de montrer comment cet état peut évoluer dans le temps (sections 4.7.2.1 et 4.7.2.2).

Définition 4.1 (Etat d'ordonnanceur)**.** *Soit l'ordonnanceur d'un système exécutant un ensemble de modules E-TDL. L'état de cet ordonnanceur est donné par l'ensemble des couples* (e_i, r_i) *où* e_i *est le temps d'exécution dont l'instance de tâche* τ_i *a besoin pour s'achever et* r_i *est le temps écoulé depuis son activation.*

Définition 4.2 (Etat d'un système)**.** *Soit un système exécutant un ensemble de modules E-TDL. A l'instant t, l'état du système* θ_t *est donné par le couple constitué de la configuration*

parallèle courante de ses modules et de l'état courant de son ordonnanceur.

Pour plus de clarté, on définit :

Para$(\theta_\mathbf{t})$ est la configuration parallèle $((m_1, \delta_1), \ldots, (m_n, \delta_n))$ dans l'état θ_t du système

Ordo$(\theta_\mathbf{t})$ est l'état de l'ordonnanceur $\{(e_1, r_1), \ldots, (e_k, r_k)\}$ dans l'état θ_t du système

Compte tenu de ces définitions, une trace d'exécution du système peut être décrite ainsi :

$$\theta_0 \rightarrow \theta_1 \ldots \rightarrow \theta_{t-1} \rightarrow \theta_t \rightarrow \theta_{t+1} \rightarrow \ldots$$

En supposant que l'état θ_{t+1} du système suive l'état θ_t du système, les configurations parallèles de ces deux états sont tout d'abord caractérisées (section 4.7.2.1), ainsi que leurs états d'ordonnanceur (section 4.7.2.2).

4.7.2.1 Evolution d'une configuration parallèle

Toute configuration parallèle est composée d'états des modules du système. On désigne par :

$(m, \delta) \in Para(\theta_t)$ l'état d'un module M à l'instant t,

$(m', \delta') \in Para(\theta_{t+1})$ l'état d'un module M à l'instant $t+1$.

Compte tenu de l'évolution d'un module E-TDL la relation suivante peut être établie entre ces deux états :

$$\begin{array}{lll} \delta' = \delta + 1 & \text{et} & m' = m \text{ si } \delta < T[m], \qquad \text{ou} \\ \delta' = 0 & \text{et} & m' \in prochains\_modes(m, \delta) \end{array}$$

Le module M peut poursuivre son exécution dans le même mode ou commencer l'exécution d'un nouveau mode si ce changement est possible. A l'instant $t+1$ il peut exister plusieurs états pouvant succéder à celui de l'instant t et satisfaisant la relation ci-dessus (par exemple, après l'évaluation d'un commutateur de mode, un module peut rester dans le mode courant ou démarrer un nouveau mode).

4.7.2.2 Evolution d'un état d'ordonnanceur

Ce sont les états successifs de l'ordonnanceur qui sont à présent examinés. Dans le cadre de cette étude la politique d'ordonnancement est *EDF* et l'ordonnanceur suit donc les règles de cette politique. Si $Ordo(\theta_t)$ est l'état de l'ordonnanceur à l'instant t, les points suivants décrivent les transformations opérées par l'ordonnanceur et déterminent son état $Ordo(\theta_{t+1})$ à l'instant $t+1$:

1. enlever les instances de tâches dont l'exécution est terminée :

$$\{\, (e_i, r_i) \,|\, (e_i, r_i) \in Ordo(\theta_t) \text{ et } e_i = 0 \,\}$$

2. incrémenter les temps écoulés depuis l'activation de chacune des instances de tâches actives :
$$\forall \tau_j : (e_j, r_j) \rightarrow (e_j, r_j + 1)$$
3. choisir une instance de tâche τ_i telle que sa date d'échéance n'est pas plus grande que la date d'échéance de toute autre instance de tâche en cours d'exécution ; réduire le temps d'exécution dont cette instance a besoin pour s'achever :
$$(e_i, r_i) \rightarrow (e_i - 1, r_i)$$
4. ajouter toutes les instances de tâches dont la date d'activation coïncide avec la configuration parallèle courante $Para(\theta_t)$:
$$\{ (C_i, 0) \mid \tau_i \text{ activée à } Para(\theta_t) \}$$

4.7.3 Simulation de l'exécution d'un ensemble de modules E-TDL

L'objectif assigné à la simulation d'un système exécutant un ensemble de modules E-TDL est de reproduire tous les états possibles de ce système afin d'en vérifier la faisabilité. La faisabilité est garantie si toute instance de tâche en cours d'exécution ne dépasse à aucun moment son échéance :
$$\forall \theta_t, (e_i, r_i) \in Ordo(\theta_t) \ : \ r_i \leq TEL_i$$
Puisque le nombre de configurations parallèles et le nombre d'états de l'ordonnanceur sont finis, le nombre d'états du système est par conséquent lui aussi fini. Il s'agit de rechercher tous ces états et de vérifier s'ils sont tous ordonnançables.
A l'instant de démarrage du système chacun de ses modules démarre l'exécution de son mode initial. Les états du système se succèdent comme décrit dans les sections 4.7.2.1 et 4.7.2.2. En cas de possible changement de mode, plusieurs états successeurs peuvent être choisis. Ces choix et, pour chacun d'eux, les séquences d'états qui leur succèdent sont à considérer. Par contre, une fois qu'un état a été pris en compte, il est inutile de le visiter une nouvelle fois quand le système le visite à nouveau. Dans ce cas, l'analyse des états successeurs est abandonnée puisque ceux-ci ont été déjà analysés. L'algorithme 16 met en œuvre ces différents points.

Le simulateur est écrit en *Python*. Afin d'optimiser les nombreuses opérations de comparaison qu'il effectue des fonctions de hachage sont utilisées.

Exemple 4.10. *Soit le système de l'Exemple 3.15. L'exemple 4.9 évalue le temps que l'outil nécessite pour une exécution du test d'ordonnançabilité de ce système. Le temps nécessaire pour réaliser la même tâche en utilisant le simulateur proposé ci-dessus est maintenant évalué. Le test est exécuté sur la même machine que celle de l'Exemple 4.9. Le simulateur a besoin de 0.983s pour vérifier l'ordonnançabilité de tous les états du système. La plus longue trace qu'il parcourt s'étend de l'instant* 0 *(démarrage du système) à l'instant* 47*.*

Algorithme 16 Vérification d'ordonnançabilité par simulation

requiert: $Modules = \{M_1, \ldots, M_n\}$ est un ensemble de n modules E-TDL
1: ordonnançable $\leftarrow$ vrai
2: $Para(\theta_0) \leftarrow ((Init[M_1], 0), \ldots, (Init[M_n], 0))$
3: $Ordo(\theta_0) \leftarrow \emptyset$
4: $new\_states_0 \leftarrow \{\, \theta_0 \,\}$
5: $visited \leftarrow \emptyset$
6: $t \leftarrow 0$
7: **pour tout** $\theta_t \in new\_states_t$ **faire**
8: **si** $\exists (e_i, r_i) \in Ordo(\theta_t) \,:\, r_i > TEL_i$ **alors**
9: ordonnançable $\leftarrow$ faux
10: **fin si**
11: **pour tout** $\zeta \in \{\ \text{TICK}(m_1, \delta_1) \times \ldots \times \text{TICK}(m_n, \delta_n) \ \mid (m_i, \delta_i) \in Para(\theta_t)\ \}$ **faire**
12: $Para(\theta_{t+1}) \leftarrow \zeta$
13: $Ordo(\theta_{t+1}) \leftarrow \text{SCHEDULE}(Ordo(\theta_t), \zeta)$
14: **si** $\theta_{t+1} \notin visisted$ **alors**
15: $visited \leftarrow visited \cup \{\theta_{t+1}\}$
16: $new\_states_{t+1} \leftarrow new\_states_{t+1} \cup \{\theta_{t+1}\}$
17: **fin si**
18: **fin pour tout**
19: $t \leftarrow t + 1$
20: **fin pour tout**
21: **renvoie** ordonnançable
22:
23: **function** TICK(m, δ)
24: **renvoie** $\{\, (m', \delta') \mid (m', \delta')$ vérifie les conditions décrites dans 4.7.2.1 $\,\}$
25: **fin function**
26:
27: **function** SCHEDULE$(taskset_t, \zeta)$
28: **renvoie** $taskset_{t+1}$ qui est produit à partir du $taskset_t$ selon les points décrits dans 4.7.2.2
29: **fin function**

4.7.4 Conclusion

Le simulateur développé, malgré sa fonctionnalité très réduite par rapport aux simulateurs cités dans la section 4.7.1, est un outil pratique pour analyser le fonctionnement d'un système modélisé en E-TDL. Il peut être utile pour la conception de nouveaux protocoles de changement de mode, la mise en œuvre de tests d'ordonnançabilité sous d'autre politique d'ordonnancement ou l'extension de ces tests à des architectures multi-processeurs. Il convient toutefois de noter, comme le montrent les Exemples 4.9 et 4.10, que le simulateur reste moins performant que l'outil.

L'utilisation du simulateur a peu d'intérêt en dehors du cadre d'E-TDL . Néanmoins, certains problèmes ouverts peuvent être identifiés. Le problème de changement de mode, abondamment étudié dans la littérature, est peu abordé par les outils de simulation. Pourtant, les tests d'ordonnançabilité en présence de changements de modes ne garantissent pas toujours une réponse exacte [2]. Il existe d'autre part peu de librairies de programmation

2. Certaines sources de surestimation dans les tests d'ordonnançabilité pour le changement de mode peuvent être limitées grâce à des techniques de simulation. A titre d'un exemple, l'interférence des tâches

facilitant le développement rapide d'outils de simulation et d'analyse d'ordonnançabilité [3]. Il serait intéressant de proposer un jeu de fonctions et de structures à partir desquelles de nouveaux outils d'analyse d'ordonnançabilité pourraient être développés. Ces différents outils ciblent différents problèmes mais ils sont basés, de manière générale, sur les mêmes concepts. Une librairie commune, avec un modèle de tâche, un modèle d'ordonnanceur, des fonctions pour le calcul de la demande ou de l'utilisation du processeurs ainsi qu'une structure basique de simulateur pourrait constituer la base du développement de futurs outils.

4.8 Conclusion

L'outil présenté dans ce chapitre permet de vérifier l'ordonnançabilité de systèmes modélisés en *E-TDL* et exécutés sous l'algorithme *EDF*. Plusieurs méthodes ont été développées pour répondre à cet objectif.

L'Algorithme 2 proposé en section 4.3 parcourt un graphe de changement de mode en évaluant les instants de début de ses modes. La traversée s'effectue selon l'algorithme de parcours en profondeur (*DFS*) en empruntant différents chemins tant que tous les instants de début de mode représenté par le nœud sont déterminés.

Les configurations parallèles d'un système sont déterminées grâce aux algorithmes exposés dans la section 4.4. Les algorithmes examinent les propriétés des instants de début de mode afin de les positionner, les uns par rapport aux autres, dans les configurations parallèles pouvant se produire réellement lors de l'exécution du système. Ces résultats sont utilisés par l'analyse exacte d'ordonnançabilité permettant de déterminer quelles sont les traces d'exécution déclenchées simultanément.

Le calcul de la demande maximale de processeur est réalisé par l'Algorithme 10 introduit dans la section 4.5. Compte tenu des changements de mode, plusieurs schémas d'exécution doivent être pris en considération pour trouver quelle est la plus grande charge dans un intervalle de durée donnée. L'algorithme applique le principe de recherche incrémentale [102] [4]. Il réutilise les résultats de ses invocations précédentes, pour des intervalles dont la durée est plus petite, afin d'accélérer le calcul dans l'intervalle courant de durée plus grande. Cette approche permet d'évaluer les demandes maximales de processeur dans tous les intervalles, dans les limites de la borne de faisabilité, plus efficacement qu'en calculant

abandonnées dans le protocole de Pedro et Burns (voir section 2.5.2.1) est évaluée par l'Equation 2.26. Dans certains cas, l'application de cette équation peut entraîner une surestimation. La valeur réelle peut seulement être obtenue par simulation.

3. Quant au développement d'un système temps réel, plusieurs librairies permettant d'alléger la tâche de leur programmation sont disponibles. On peut citer en particulier *ptask* [41]. Cette librairie fait abstraction des opérations de bas niveau et offre des structures pour la programmation des systèmes temps réel. Elle permet également d'opérer des changements de modes.

4. Ce type de recherche est souvent utilisé pour résoudre des problèmes dont l'une des contraintes change avec le temps ainsi que pour résoudre des problèmes de cheminement.

celles-ci sans se fonder sur les résultats précédents. Wandeler [177], pour déterminer le nombre d'occurrences d'un événement dans un intervalle, utilise une approche similaire (poids moyen minimal d'un cycle [95]).

Le fonctionnement du test d'ordonnançabilité pour *E-TDL* sous l'algorithme *EDF* est décrit dans la section 4.6. Sa complexité est réduite en combinant deux type d'analyse : approchée et exacte. Si, pour une durée d'intervalles donnée, l'analyse approchée ne permet pas de garantir l'ordonnançabilité du système, les intervalles sont examinés par l'analyse exacte. Dans l'Exemple 4.9, l'analyse exacte est nécessaire seulement pour deux des 54 durées d'intervalles considérées.

Quant au système utilisé dans les exemples présentés dans ce chapitre (pour sa définition voir Exemple 3.15), sa simplicité, parfaite pour illustrer les détails de l'analyse, ne permet pas d'évaluer objectivement la performance du test. Sans avoir exécuté d'expérimentations avec des jeux de systèmes plus complets et plus exhaustifs, il est difficile d'estimer la taille maximale permettant d'obtenir des résultats d'analyse dans des délais raisonnables.

Comme montré au travers des Exemples 4.9 et 4.10, l'outil requiert moins de temps pour vérifier l'ordonnançabilité que le simulateur (section 4.7). De plus, il serait possible d'obtenir une meilleure performance par intégration de techniques d'approximation de la demande de processeur dans l'outil. Cette amélioration, ainsi que d'autres liées à l'extension du langage, sont discutées dans le chapitre suivant.

Chapitre 5

Conclusions et perspectives

Plusieurs aspects des travaux présentés dans cette thèse sont susceptibles d'améliorations futures. Ces dernières s'articulent autour de la définition du langage (section 5.1) et de l'analyse d'ordonnançabilité (section 5.2). Toutefois, les travaux les plus essentiels à réaliser consistent à proposer un outil complet permettant de concevoir les systèmes temps-réel, de vérifier leur ordonnançabilité et d'en assurer une exécution correcte. Ces travaux pourraient se poursuivre dans les directions suivantes :

- analyse syntaxique du langage
- environnements de développement pour les systèmes temps-réel
- compilation du langage pour produire un schéma du contrôle temporel de l'exécution
- développement du micro noyau temps réel (nouvelles architectures et fonctionnalités)

5.1 Extensions du langage

Le protocole de changement de mode dans *E-TDL* se fonde sur le protocole défini par son prédécesseur *TDL*. La section 5.1.1 montre que le modèle de tâche proposé par *E-TDL* permet de définir des protocoles plus réactifs. Il est de plus envisageable de préparer à l'avance le changement de mode par une observation anticipée de ses conditions. Les questions relatives à l'introduction de contraintes sur le nombre d'exécutions d'un mode et à leur impact sur l'analyse d'ordonnançabilité font l'objet de la section 5.1.2. Une discussion sur les extensions possibles du modèle de tâche dans *E-TDL* est exposée en section 5.1.3.

5.1.1 Proposition de deux nouveaux protocoles pour le changement de mode dans E-TDL

Cette section propose deux nouveaux protocoles de changement de mode pour *E-TDL*. Le premier protocole (section 5.1.1.1) exploite le modèle de tâche défini dans *E-TDL*. Le second protocole (section 5.1.1.2), inspiré du protocole du délai minimal (*MSO*), introduit

un nouveau mécanisme de changement de mode jamais mis en place dans les langages de définition temporelle.

5.1.1.1 Changement de mode anticipé

Les langages de définition temporelle imposent que l'exécution logique d'une tâche d'un mode courant ne soit jamais interrompue par un changement de mode (voir section 2.6). Dans *TDL*, les seuls instants où un changement de mode peut intervenir sont les hyper-périodes de mode. Toutes les tâches d'un mode sont alors logiquement inactives. Le même mécanisme du changement de mode est adopté par *E-TDL*. Toutefois, le nouveau modèle de tâche introduit (voir section 1.5.2) permet d'assouplir ces règles. Une tâche *E-TDL* τ_i d'un mode m est logiquement inactive dans les intervalles suivants :

$$inactive(\tau_i) = \{\, [jT_i, jT_i + \phi_i], [jT_i + \phi_i + TEL_i, (j+1)T_i] \mid j \in \mathbb{N},\ j < T[m]/T_i \,\} \quad (5.1)$$

Les plages temporelle où aucune de tâche du mode m n'est active sont définies comme :

$$\bigcap_{\tau_i \in \tau[m]} inactive(\tau_i) \quad (5.2)$$

En identifiant ces plages temporelles il serait possible de changer de mode d'exécution sans attendre la fin d'hyperpériode du mode [98]. Un protocole similaire a été proposé pour *Giotto* par Martinek et Pohlmann [114] (voir section 2.6.1).

Exemple 5.1. *Soit un module E-TDL pouvant exécuter à la fois un des deux modes : A ou B. Dans le mode A s'exécutent les tâches* $\tau_1 = (0, 2, 8, 12)$ *et* $\tau_2 = (0, 5, 10, 18)$. *Dans le mode B s'exécutent les tâches* $\tau'_1 = (0, 4, 14, 16)$ *et* $\tau'_2 = (0, 2, 14, 16)$. *Les points rouges*

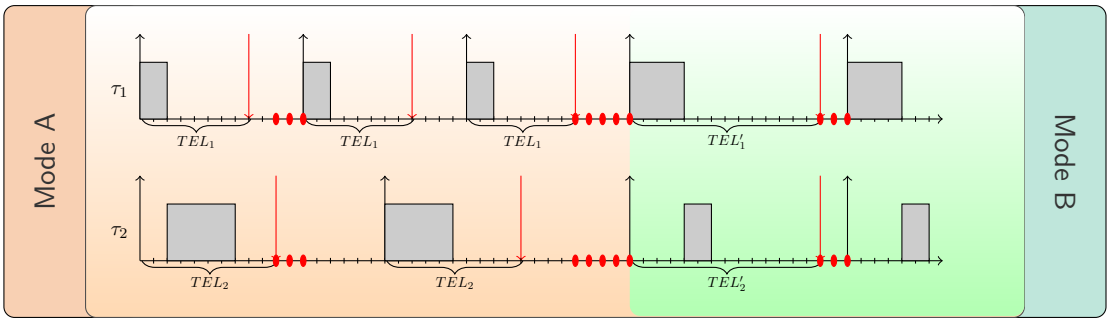


FIGURE 5.1 – Temps de mode admissibles pour un changement de mode selon les règles assouplies

sur la Figure 5.1 indiquent les temps de mode où aucune tâche du mode courant n'est logiquement active et où un changement de mode peut s'opérer (selon les règles assouplies de changement de mode).

La mise en œuvre de cette nouvelle règle de changement de mode affecterait l'analyse d'ordonnançabilité présentée au sein des chapitres précédents. Les évaluations de commutateur de mode ne seraient plus seulement périodiques. La structure du graphe de changement de mode (voir section 3.2.1) devrait pendre cela en compte. Ses arcs devraient permettre d'exprimer qu'un changement de mode peut se produire dans un intervalle de temps donné.

Exemple 5.2. *Soit le même module que celui de l'Exemple 5.1. En tenant compte de la Formule 5.2, un changement de mode dans le mode A peut être déclaré à un temps de mode* $\delta_{sw}^{A} \in [10, 12] \cup [32, 36]$. *Dans le mode B un changement de mode peut être déclaré à un temps de mode* $\delta_{sw}^{B} \in [14, 16]$. *La Figure 5.2 représente un graphe de changement de mode pour le module si on considère que le mode A peut être activé à tout temps* δ_{sw}^{B} *de mode B et le mode B à tout temps* δ_{sw}^{A} *de mode A.*

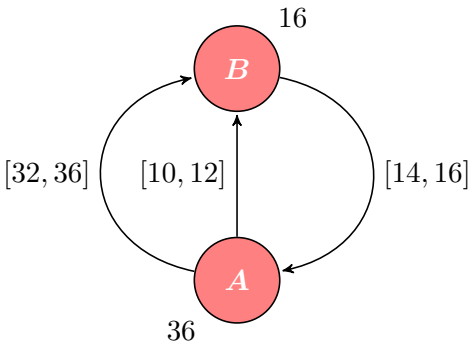


FIGURE 5.2 – Graphe de changement de mode pour les règles de changement de mode assouplies

La méthode du calcul des instants auxquels débutent les modes (voir section 3.6.1) devrait par conséquent être réexaminée. Cela entraîne aussi des modifications dans l'établissement des relations entre instants de début de mode dans des modules différents (section 3.6.2), dans l'algorithme du parcours du graphe de changement de mode (section 4.3) et dans le calcul de la demande de processeur (section 4.5).

De nouveaux points pouvant permettre un changement de mode pourraient être exhibés en admettant que l'exécution de certaines tâches puisse être abandonnée. Cela pose, entre autres, un problème dans l'estimation exacte de la quantité de temps de processeur utilisée par les tâches avant l'abandon de leur exécution. Pour y remédier, la même méthode que celle proposée dans le protocole de Pedro et Burns (voir section 2.5.2.1 et Formule 2.26) pourrait être utilisée [1].

1. Cette approche a été déjà appliquée pour *EDF* par Fisher et Ahmed [63].

5.1.1.2 Proposition d'un protocole de délai minimal pour E-TDL

Si un commutateur de mode observe qu'à un instant donné les conditions d'un changement de mode sont réunies, le changement est immédiatement effectué à cet instant. En distinguant l'observation des conditions de changement de mode et le changement de mode lui-même, la nécessité d'un changement pourrait être détectée à l'avance et le système, de façon dynamique, pourrait modifier son comportement afin de réaliser le changement de mode dans le plus court délai possible. Cela plaide pour adopter dans *E-TDL* le protocole du délai minimal (*MSO*) décrit en section 2.5.1.2.

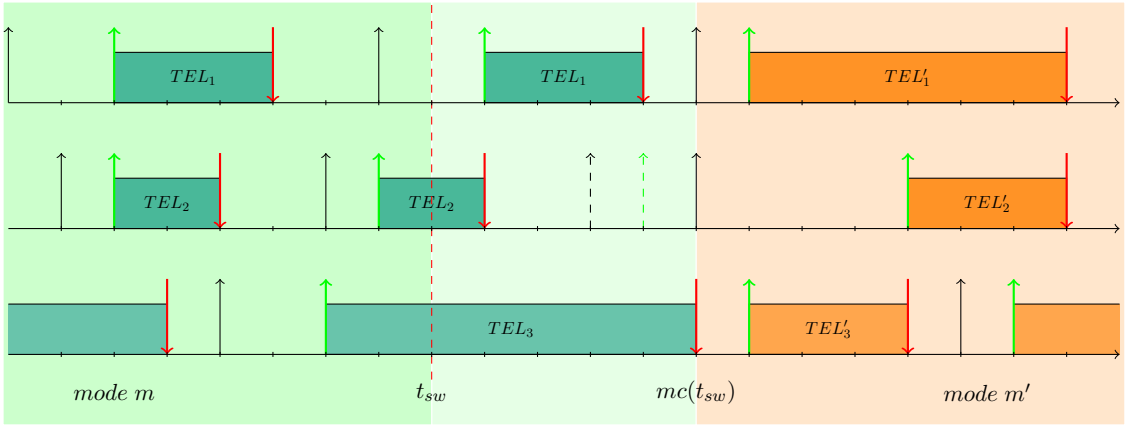


FIGURE 5.3 – Protocole de délai minimal pour E-TDL

Certaines modifications doivent cependant être apportées au protocole pour assurer sa conformité au paradigme du temps d'exécution logique. Le changement de mode, dans la version du protocole proposée pour *E-TDL*, se déroule ainsi. A partir de l'instant de vérification d'un changement de mode δ_{sw}, si les conditions de ce changement sont réunies, le schéma d'activation des tâches dans le mode courant m est modifié. Aucune instance de tâche dont l'échéance dépasserait la plus grande échéance d'une instance de tâche s'exécutant à l'instant δ_{sw} n'est activée. Ainsi, après un certain délai, toutes les tâches du mode courant sont achevées. La formule qui suit permet de calculer l'instant $mc(\delta_{sw})$ du changement de mode dont les conditions sont vérifiées à l'instant δ_{sw}.

$$mc(\delta_{sw}) = \max_{\tau_i \in \tau[m]} \left\{ \max \left(\left\lfloor \frac{\delta_{sw} - \Phi_i}{T_i} \right\rfloor T_i + \Phi_i + TEL_i,\ \delta_{sw} \right) \right\} \tag{5.3}$$

Lemme 5.1. *Le plus grand délai Δ_{mc} séparant l'instant de vérification des conditions de changement de mode δ_{sw} et l'instant auquel ce changement est effectué $mc(\delta_{sw})$ dans un mode m est égal au plus grand Temps d'Exécution Logique des tâches de ce mode.*

Démonstration. Par définition, $\Delta_{mc} = mc(\delta_{sw}) - \delta_{sw}$. Soit τ_i la tâche du mode m pour laquelle la valeur de Δ_{mc} est la plus grande. Soit $d_{i,j}$ la date d'échéance de la plus récente instance de τ_i activée avant ou à l'instant δ_{sw}. On remarque que :

$$d_{i,j} = \left\lfloor \frac{\delta_{sw} - \Phi_i}{T_i} \right\rfloor T_i + \Phi_i + TEL_i$$

Deux cas sont possibles. Si $\delta_{sw} \in [d_{i,j}, r_{i,j+1}]$ alors $mc(\delta_{sw}) = \delta_{sw}$ et $\Delta_{mc} = 0$. Autrement, $\delta_{sw} \in [r_{i,j}, d_{i,j}]$ alors $mc(\delta_{sw}) = d_{i,j}$ et $\Delta_{mc} = d_{i,j} - \delta_{sw}$. La valeur de Δ_{mc} est la plus grande pour $\delta_{sw} = r_{i,j}$. Alors $\Delta_{mc} = d_{i,j} - r_{i,j} = TEL_i$. Puisque le délai Δ_{mc} est le plus grand pour τ_i, $\forall \tau_j \in \tau[m] : TEL_i \geq TEL_j$. □

Exemple 5.3. *Considérons un mode d'E-TDL exécutant les tâches suivantes :* $\tau_1 = (3, C_1, 6, 10)$, $\tau_2 = (1, C_2, 3, 8)$ *et* $\tau_3 = (0, C_3, 5, 5)$. *Leurs pires temps d'exécution,* C_1, C_2 *et* C_3, *sont choisis de telle façon que le mode soit faisable. Supposons que le changement de mode puisse s'opérer de la manière décrite ci-dessus. La Figure 5.4(a) montre la valeur de* $mc(\delta_{sw})$ *en fonction de l'instant* δ_{sw} *(voir Equation 5.3) pour le mode. La Figure 5.4(b) montre le délai* Δ_{mc} *entre l'instant de vérification des conditions de changement de mode* δ_{sw} *et l'instant auquel ce changement est effectué dans le mode.*

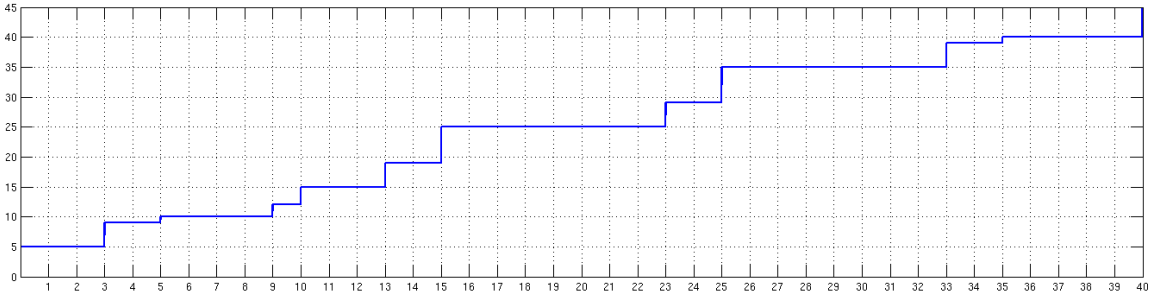
5.1.2 Expression de contraintes sur le nombre d'exécutions d'un mode

Plusieurs travaux [138, 139, 63] limitent inférieurement ou supérieurement le nombre d'exécutions d'un mode. Ce nombre pourrait aussi être limité dans *E-TDL*. Un module pourrait rester dans un mode tant qu'il ne l'exécute pas au-delà d'un nombre maximal de fois (une infinité si ce nombre n'est pas spécifié). Il pourrait le quitter après l'avoir exécuté au moins un nombre minimal de fois (une fois si ce nombre n'est pas spécifié).

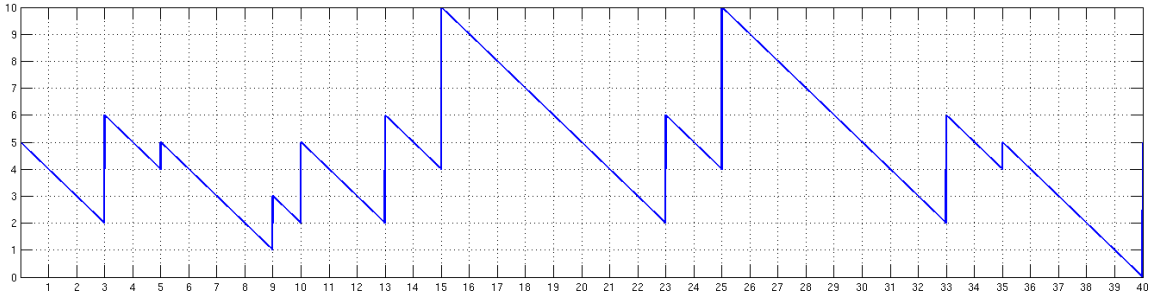
L'information sur le temps minimal et maximal d'exécution d'un mode permet de réduire le pessimisme de l'analyse d'ordonnançabilité. L'évaluation de la charge du processeur peut donner des résultats inférieurs à ceux obtenus en considérant qu'un mode puisse s'exécuter infiniment. De plus, la borne de faisabilité peut être évaluée plus précisément.

Des relations logiques conditionnant l'exécution des modes pourraient rendre l'analyse d'ordonnançabilité moins pessimiste. Si un mode est activé lorsqu'une condition est vraie et si un autre mode, dans module différent, l'est lorsque celle-ci est fausse, ces deux modes ne seront jamais activés simultanément. Les charges produites par leur exécution simultanée ne devraient pas être sommées en évaluant la charge globale du système.

La charge courante du système peut être utilisée dans les conditions de changement de mode [8]. Un mode peut fournir une fonctionnalité qu'un autre mode peut réaliser avec une qualité moindre et avec moins de ressources processeur. Le système peut choisir lequel de ces deux mode exécuter en fonction de la charge de modes plus prioritaires exécutés dans d'autres modules. Le comportement observable devient alors dépendant des ressources de la plateforme d'exécution.



(a) Instant de changement de mode $mc(\delta_{sw})$ en fonction du temps de mode δ_{sw}



(b) Valeur du délai Δ_{mc} en fonction du temps de mode δ_{sw}

FIGURE 5.4 – Caractéristique des changements de mode opérés selon le protocole du délai minimal pour E-TDL dans le mode de l'Exemple 5.3

5.1.3 Revisiter le Temps d'Exécution Logique

La sémantique *d'E-TDL* impose que la $j^{ème}$ instance ($j \in \mathbb{N}_+$) d'une tâche τ_i activée dans l'intervalle de temps $[(j-1)T_i, jT_i]$ soit terminée avant l'instant jT_i (voir section 3.1). La plus grande valeur de TEL_i de la tâche τ_i qui satisfait La Formule 3.2 est égale à :

$$TEL_i = T_i - \Phi_i \tag{5.4}$$

La prochaine instance de τ_i doit donc être activée à l'instant $jT_i + \Phi$. L'intervalle $[jT_i, jT_i + \Phi_i]$ ne peut cependant pas être ouvert à l'exécution de l'instance courante de τ_i. Si cette instance pouvait poursuivre son exécution logique dans cet intervalle, l'instance suivante démarrerait après sa terminaison. La plus grande valeur de Temps d'Exécution Logique qui pourrait toujours alors être attribué à la tâche τ_i serait égale à :

$$TEL_i = T_i \tag{5.5}$$

La question sur la levée de la contrainte définie par la Formule 3.2 peut donc se poser. Le modèle de tâche *d'E-TDL*, en l'absence de cette contrainte, est présenté sur la Figure 5.5.

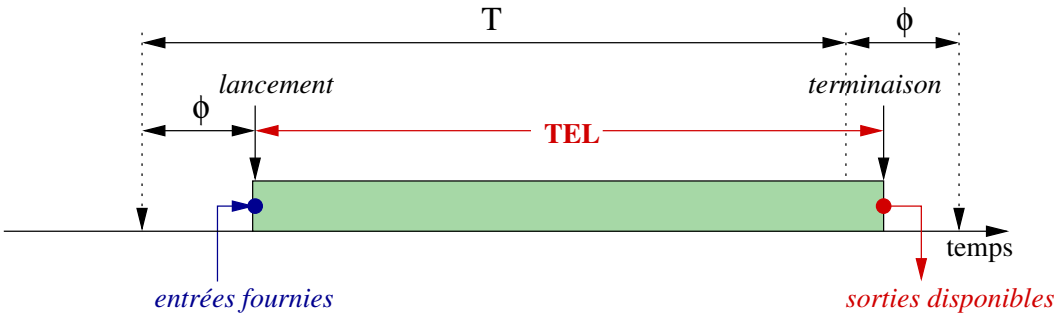


FIGURE 5.5 – Temps d'Exécution Logique tel que $TEL_i \leq T_i$

Ce modèle, amplement utilisé dans le contexte de tâches temps-réel classiques, n'est pas sans ambiguïté dans le contexte de tâches dirigées par le temps. Il permet en effet qu'à tout temps d'un mode (y compris aux instants correspondant à son hyperpériode) une tâche puisse être en cours d'exécution logique [2]. Un nouveau mécanisme de changement de mode devrait alors être défini et appliqué (voir par exemple section 5.1.1.2).

Le modèle de tâche en *E-TDL* peut aussi être doté de nouveaux paramètres. Derler [56] permet par exemple qu'une tâche lise et écrive des données en dehors des bornes de son Temps d'Exécution Logique si cela ne modifie pas le comportement observable du système. La lecture des entrées ou l'écriture des sorties d'une tâche pourraient être déclarées pouvoir se réaliser au sein d'un intervalle. Une tâche disposerait d'un délai J_{start} pour lire ses entrées et d'un autre délai J_{stop} pour écrire ses sorties. Le système aurait ainsi plusieurs comportements observables.

Dans *xGiotto* [72] ainsi que dans *TDL* depuis la version 1.5 [1], des tâches du système peuvent être déclenchées par des événements. Ce type de tâches n'est pas spécifié dans *E-TDL*. Cependant, les modules *d'E-TDL* peuvent s'exécuter en présence d'autres composants, y compris ceux à déclenchement événementiel. L'analyse d'ordonnançabilité en tient compte.

5.2 Améliorations possibles de l'analyse

Plusieurs voies pourraient être explorées pour améliorer et optimiser les analyses précédemment décrites et proposées.

Les protocoles de changement de mode proposés en section 5.1.1 imposent une nouvelle structure du graphe de changement de mode. Les instants de début de mode pourraient être examinés par des méthodes tenant compte du fait qu'un changement de mode peut survenir au delà des seuls instants des multiples de l'hyperpériode (section 5.2.1).

2. Par exemple, un mode *E-TDL* exécutant deux tâche $\tau_1 = (0, 1, 2, 2)$ et $\tau_2 = (1, 1, 2, 2)$.

Plusieurs aspects du test d'ordonnançabilité présenté dans ce manuscrit nécessiteraient des enrichissements et des améliorations. Le test est basé sur la fonction de demande. Le temps nécessaire pour évaluer la demande maximale de processeur pourrait être réduit en caractérisant la cyclicité de sa croissance (section 5.2.4) ou en approximant sa valeur par des méthodes dont le calcul est moins complexe (section 5.2.3). En affinant la borne de faisabilité (section 5.2.5), le nombre d'intervalles dans lesquels la demande de processeur est évaluée pourrait être réduit. L'analyse d'ordonnançabilité d'un système pourrait tenir compte des étapes successives de son développement et réutiliser des résultats antérieurs (section 5.2.6). Le test pourrait aussi se fonder sur une approche basée sur l'analyse de période d'activité du processeur (section 5.2.2) plutôt que sur la fonction de demande.

Dans le prolongement de ces travaux, il paraît important de proposer des tests d'ordonnançabilité pour les algorithmes à priorité fixe (section 5.2.7) ainsi que pour des architectures multiprocesseurs (section 5.2.9). Le choix de meilleurs modèles d'allocation de ressources pour des systèmes dirigés par le temps (section 5.2.8) doit aussi être examiné.

5.2.1 Extension des algorithmes de parcours des graphes de changement de mode pour l'introduction de nouveaux protocoles de changement de mode

Le changement de mode, dans les protocoles proposés en section 5.1.1, peut se produire à un instant autre qu'un multiple de l'hyperpériode du mode. La structure du graphe de changement de mode dotée d'arcs représentant ce nouveau type de transitions nécessite une nouvelle analyse. Son objectif est de caractériser les instants de début de chaque mode dans le graphe de changement de mode.

On suppose que la condition du changement de mode d'un mode m_k vers un mode m_{k+1} est vérifiée à tout instant $\mu_k \cdot T_{sw}(m_k, m_{k+1}) + \varphi_k$ dans m_k après sa première activation ($\mu_k, \varphi_k \in \mathbb{N}$ et $0 \leq \varphi_k < T_{sw}(m_k, m_{k+1})$). Une transition dans le graphe de changement de mode (voir Définition 3.1) est alors décrite par le triplet $(m_k, m_{k+1}, T_{sw}(m_k, m_{k+1}) + \varphi_k)$. L'ensemble des instants de début d'un mode est donné par les longueurs des chemins allant du mode initial jusqu'à ce mode. Compte tenu du délai φ_k introduit dans les transitions, la longueur d'un chemin w (voir Formule 3.6) traversant le graphe de changement de mode s'exprime comme :

$$|w| = \sum_{k=1}^{n} \mu_k \cdot T_{sw}(m_k, m_{k+1}) \; + \; \varphi_k \tag{5.6}$$

Or il s'agit de trouver la séquence des instants auxquels un mode donné peut commencer son exécution. Cette séquence correspond à des longueurs de chemins conduisant à ce mode. Une chaîne de base (voir Définition 3.14) est une représentation de chemins passant par les même modes. Considérons deux cas :

a) le chemin w est acyclique

La longueur du chemin w est donnée par :

$$|w| = \sum_{k=1}^{n} \mu_k \cdot T_{sw}(m_k, m_{k+1}) \; + \; \varphi_k = \sum_{k=1}^{n} \mu_k \cdot T_{sw}(m_k, m_{k+1}) \; + \; \sum_{k=1}^{n} \varphi_k$$

Soit p une chaîne correspondant au chemin w. Selon l'Equation 3.32 :

$$\sum_{k=1}^{n} \mu_k \cdot T_{sw}(m_k, m_{k+1}) = pgcd(p) \cdot \sum_{k=1}^{n} \mu_k \cdot T'_{sw}(m_k, m_{k+1})$$

En remplaçant $\sum_{k=1}^{n} \varphi_k$ par :

$$\sum_{k=1}^{n} \varphi_k = pgcd(p) \cdot \left\lfloor \frac{\sum_{k=1}^{n} \varphi_k}{pgcd(p)} \right\rfloor + \left(\sum_{k=1}^{n} \varphi_k \right) \bmod pgcd(p)$$

On obtient :

$$|w| = pgcd(p) \cdot \left(\sum_{k=1}^{n} \mu_k \cdot T'_{sw}(m_k, m_{k+1}) + \left\lfloor \frac{\sum_{k=1}^{n} \varphi_k}{pgcd(p)} \right\rfloor \right) + \left(\sum_{k=1}^{n} \varphi_k \right) \bmod pgcd(p)$$

La longueur de tout chemin w suivant la chaîne p appartient à l'ensemble suivant :

$$|w| \in \left\{ n \cdot pgcd(p) + \left(\sum_{m_k \in p} \varphi_k \right) \bmod pgcd(p) \,|\, n \in \mathbb{N} \right\} \tag{5.7}$$

b) le chemin w contient un cycle

On désigne par w_c le chemin dont les transitions ne se répètent pas et constituent le cycle dans le chemin w. On désigne par w_s le chemin simple (*un chemin simple* ne passe pas deux fois par une même arête) conduisant depuis le mode initial jusqu'au mode $tail(w)$ du chemin w. Le chemin w suit le chemin simple w_s en passant un certain nombre de fois $n_c \in \mathbb{N}$ par le cycle w_c.

$$\begin{aligned} |w| &= |w_s| + n_c \cdot |w_c| = \\ &= \sum_{m_k \in w_s} \mu_k \cdot T_{sw}(m_k, m_{k+1}) + \varphi_k \;\; + \;\; n_c \cdot \sum_{m_k \in w_c} \mu_k \cdot T_{sw}(m_k, m_{k+1}) + \varphi_k \end{aligned}$$

Si w_s correspond à une chaîne $p_s \subseteq p$ et w_c à une chaîne $p_c \subseteq p$, la longueur du chemin w s'exprime, pour $n_1, n_2 \in \mathbb{N}$, comme :

$$|w| = \; n_1 \cdot pgcd(p_s) + \left(\sum_{m_k \in p_s} \varphi_k \right) \bmod pgcd(p_s) + n_c \cdot n_2 \cdot pgcd(p_c) + n_c \cdot \sum_{m_k \in p_c} \varphi_k$$

Pour $n \in \mathbb{N}$ suffisamment grand (voir le problème de Frobenius en section 3.6.1) :

$$n_1 \cdot pgcd(p_s) + n_2 \cdot pgcd(p_c) = n \cdot pgcd(p_s \cup p_c) = n \cdot pgcd(p)$$

La longueur de tout chemin w suivant la chaîne p composée des chaînes p_s et p_c appartient à l'ensemble suivant :

$$|w| \in \left\{ n \cdot pgcd\left(p \cup \sum_{m_k \in p_c} \varphi_k\right) + \left(\sum_{m_k \in p_s} \varphi_k\right) \bmod pgcd(p_s) \,|\, n \in \mathbb{N} \right\} \tag{5.8}$$

Dans certains cas, pour un chemin w plusieurs chemins simples peuvent être distingués.

Exemple 5.4. *Soit un graphe de changement de mode tel que celui de la Figure 5.6. Puisque le graphe est acyclique, la longueur de tout chemin w depuis le mode m_1 via le mode m_2 jusqu'au mode m_3 peut être obtenue grâce à la Formule 5.7. La chaîne p correspondant au chemin w est $p = \{T_{sw}(m_1, m_2), T_{sw}(m_2, m_3)\}$. L'expression $\sum_{m_k \in p} \varphi_k$ est égale à $\varphi_1 + \varphi_2$.*

Soit maintenant le graphe de changement de mode de la Figure 5.7. Ce graphe est obtenu en reliant par une arête les modes m_3 et m_2 du graphe précédent. Ainsi, un cycle est créé. La longueur de tout chemin w de m_1 jusqu'à m_3 en traversant plusieurs fois les modes m_2 et m_3 du cycle est donnée par la Formule 5.8. La seule chaîne simple p_s dans le graphe passe par les modes m_1, m_2 et m_3 : $p_s = \{ T_{sw}(m_1, m_2), T_{sw}(m_2, m_3) \}$. La chaîne du cycle p_c traverse les modes m_2 et m_3 : $p_c = \{ T_{sw}(m_3, m_2), T_{sw}(m_2, m_3) \}$. L'expression $\sum_{m_k \in p_c} \varphi_k$ est égale à $\varphi_3 + \varphi_2$ et l'expression $\sum_{m_k \in p_s} \varphi_k$ à $\varphi_1 + \varphi_2$.

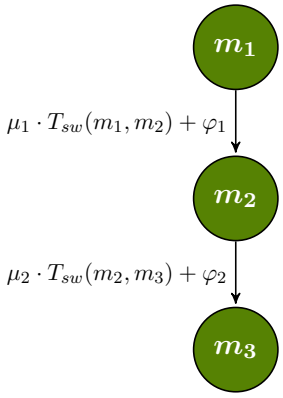


FIGURE 5.6 – Chemin sans cycle

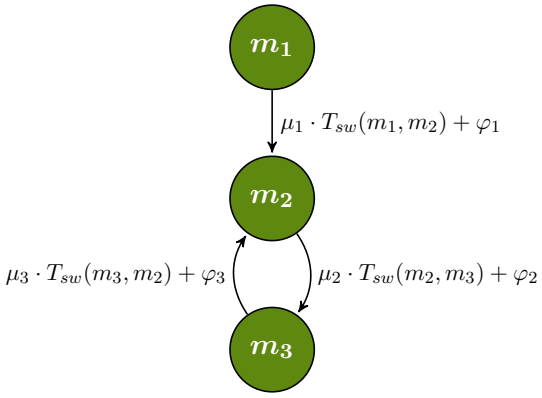


FIGURE 5.7 – Chemin avec cycle

5.2.2 Analyse basée sur les périodes d'activité

Le test d'ordonnançabilité *d'E-TDL* sous *EDF* est basé sur le calcul d'une fonction de demande. Pourtant, la méthode d'analyse des périodes d'activité du processeur (*busy*

period) [70, 38] présentée en section 2.4.2 (Les Formules 2.18 et 2.19) pourrait être également utilisée. On trouve de nombreux tests utilisant cette méthode dans la littérature [132, 160, 148]. L'analyse des périodes d'activité du processeur permet de limiter la durée des intervalles à vérifier lors de l'analyse d'ordonnançabilité. Contrairement à la fonction de demande, cette approche ne permet pas d'extraire d'information sur le temps de processeur disponible dans un intervalle donné. Une analyse des périodes d'activité du processeur serait envisageable si l'ensemble des modules *E-TDL* s'exécutait sur un processeur exclusivement.

5.2.3 Simplification de l'analyse par exploitation de la périodicité de la fonction de la demande maximale

S'il était possible de démontrer que la fonction de demande maximale $maxdf_M$ devient périodique au bout d'intervalle donné, la méthode de calcul serait plus efficace. La fonction $maxdf_M$ est *presque périodique* s'il existe $p, q \in \mathbb{N}$ tels qu'à partir de Δ_{min} :

$$\forall\, \Delta > \Delta_{min} : maxdf_M(m, \delta, \Delta + p) = maxdf_M(m, \delta, \Delta) + p \cdot q \tag{5.9}$$

Il existe une ressemblance entre *graphe de changement de mode* et *digraph* [166]. Un *digraph* est un modèle de tâche temps-réel basé sur un graphe orienté. Les sommets de ce graphe sont des sous-tâches et les arc définissent leur ordre d'exécution ainsi que les intervalles de temps séparant les exécutions de sous-tâches adjacentes. Zeng et Di Natale ont prouvé que la fonction *dbf* est *presque périodique* pour tout *digraph* [178, 179]. Ils proposent une procédure permettant de déterminer les paramètres périodiques de cette fonction [3]. Elle s'applique uniquement dans le cas de graphes fortement connexes.

Il n'est pas requis que le *graphe de changement de mode* soit fortement connexe. Cependant, pour les graphes satisfaisant cette propriété l'application de cette méthode pourrait permettre de simplifier le calcul de la fonction de demande maximale. Il existe également des cas où la périodicité de la fonction de demande maximale est encore plus évidente : module composé d'un seul mode (voir le module M_3 de l'Exemple 3.15), module dont tous les modes ont la même période et dont le graphe de changement de mode est complet (tous les modes sont reliés entre-eux).

5.2.4 Approximation de la demande maximale

La fonction de demande associée à une trace de module *E-TDL* (Définition 3.9) est composée comme la somme de deux fonctions : *fonction de demande dans un mode* (Défi-

3. Un graphe est représenté par la matrice des distances entre ses différents sommets. La distance maximale entre deux sommets, via un nombre de sommets donné, correspond à une série de produits matriciels dans la structure d'algèbre *max-plus* [19]. Cette distance est associée à la demande maximale de temps processeur entre les activations des deux sous-tâches dans un *digraph*. Dans une structure *max-plus* les éléments d'une matrice obtenue comme une série de produits matriciels d'elle-même peuvent être calculés par une fonction presque périodique. Cette propriété est utilisée pour trouver les paramètres périodiques de la fonction *dbf* des *digraphs*.

nition 3.7) et *fonction de demande d'un chemin* (Définition 3.8). En bornant la valeur de chacune de ces fonctions la fonction de demande maximale peut être approximée.

La fonction de demande dans un mode détermine la demande faite pendant l'exécution du premier et du dernier mode d'une trace de module. La fonction évalue la demande d'un ensemble de tâches qui ne change pas. Les méthodes du calcul de la fonction de demande pour les ensembles de tâches uniformes (sans changement de mode) et leurs approximations sont discutées en section 2.4.2. La complexité du calcul de la fonction de demande dans un mode peut en particulier être réduite en considérant que le départ de toutes les tâches est simultané. Le coût de ce calcul peut encore être réduit par application d'autres méthodes d'approximation [44, 5, 6]. Si ces approximations sont trop pessimistes le test de Pellizzoni et Lipari pour les tâches asynchrones peut être utilisé. Dans le cas du test pour un système unimodal (section 3.4) le nombre des intervalles à vérifier peut être réduit en appliquant le test *QPA* (*Quick Processor-Demand Analysis*) [180, 181].

La fonction de demande d'un chemin w évalue le temps processeur nécessaire pour l'exécution complète des modes de w. La valeur de la fonction est calculée avec l'Equation 3.9. En supposant la durée du chemin w (Equation 3.6) égale à Δ :

$$\Delta \quad = \sum_{(m_k,\mu_k)\in w} \mu_k \cdot T_{sw}(m_k, m_{k+1}) \tag{5.10}$$

Soit U_{max} le plus grand taux d'utilisation d'un mode du module exécutant le chemin :

$$\forall (m_k, \mu_k) \in w \ : \ U(m_k) \ \leq \ U_{max} \tag{5.11}$$

Pour tout chemin w de durée Δ :

$$df(w) \ \leq \ \Delta \cdot U_{max} \tag{5.12}$$

Il est possible de trouver une borne moins pessimiste. La borne précédente est obtenue en supposant que le mode dont le taux d'utilisation est le plus élevé puisse s'exécuter entièrement un certain nombre de fois dans l'intervalle Δ. Cela est vrai si Δ est un multiple de l'hyperpériode du mode. Dans le cas contraire, une borne plus précise peut être obtenue en supposant que la séquence d'exécution des modes est telle que sa durée ne dépasse pas Δ. Il s'agit donc de trouver une séquence d'exécution des modes dont la demande de temps processeur est maximale dans un intervalle de durée Δ. Ce problème correspond à celui *du sac à dos* (*Knapsack problem*).

$$\text{maximiser} \ : \qquad \sum_k \mu_k \cdot H(m_k) \cdot U(m_k) \tag{5.13}$$

$$\text{avec} \ : \qquad \sum_k \mu_k \cdot H(m_k) \leq \Delta \tag{5.14}$$

$$\forall k : \mu_k \in \mathbb{N},\ m_k \in Modes[M] \tag{5.15}$$

Soit $full(\Delta)$ la valeur de la Formule 5.13 pour Δ. Si $full(\Delta)$ est évalué pour tous les intervalles Δ dans un ordre croissant, comme cela est réalisé dans le test d'ordonnançabilité proposé au chapitre précédent, une technique de programmation dynamique peut être utilisée pour déterminer sa valeur :

$$full(\Delta) = \max_k \{ full(\Delta - H(m_k)) + U(m_k) \} \tag{5.16}$$

Exemple 5.5. *Soit un module pouvant exécuter à la fois un des deux modes :* m_1 *ou* m_2*. Les modes sont caractérisés par les hyperpériodes et les taux d'utilisation suivants :* $H(m_1) = 3$*,* $H(m_2) = 4$ *et* $U(m_1) = 1$*,* $U(m_2) = 2$*. La Figure 5.8 représente la fonction* $full(\Delta)$ *pour le module.*

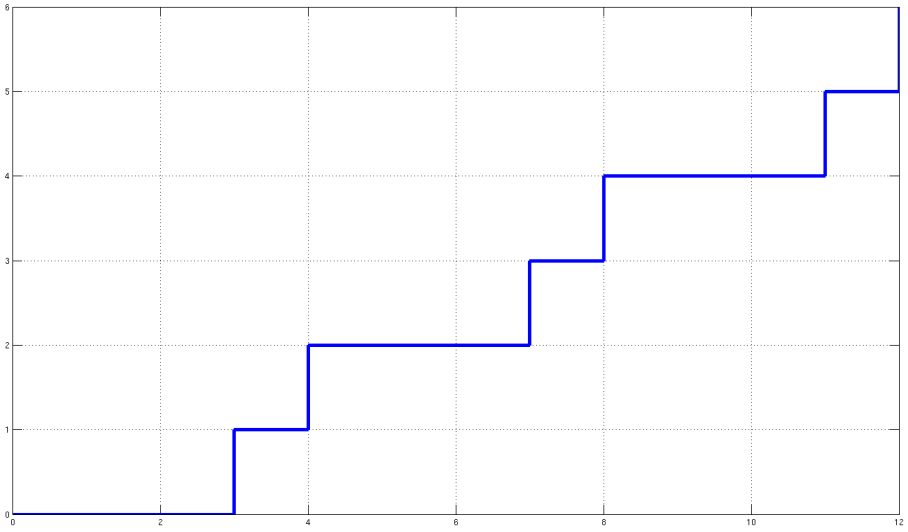


FIGURE 5.8 – Fonction $full(\Delta)$ pour le module de l'Exemple 5.5

Baruah [26] propose une solution similaire pour le calcul de la demande dans le modèle des multi-trames généralisées (*Generalized Multiframe*) non-cycliques [118, 119]. La méthode proposée ici ne tient cependant pas compte de l'ordre dans lequel les modes peuvent s'exécuter. Par conséquent, certaines séquences d'exécution des modes sont surestimées. Afin, le cas échéant, de réduire cette surestimation, la méthode pourrait composer les séquences en tenant compte des relations de précédence entre modes puis revenir à la méthode moins précise dès qu'elle suffit à garantir l'ordonnançabilité.

5.2.5 Précision de la borne de faisabilité

Une borne de faisabilité plus précise pourrait être obtenue pour chaque configuration parallèle ζ, dans les Formules 3.19 et 3.20, en utilisant la fonction $maxdf_{M_k}(m_k, \delta_k, \Delta)$ au lieu de la fonction $maxdf_{M_k}(\Delta)$. Cette borne de faisabilité serait donc calculée en fonction de ζ. Dans l'Expression 3.19, la fonction $df_M(\Delta_1)$ peut être bornée par :

$$df_{M_k}(\Delta_1) \ \leq \ \max_{m_k} df_{M_k}(m_k, r_{\min}(m_k), H[m_k])$$

pour $(m_k, \delta_k) \in \zeta$. La fonction $df_M(\Delta_2)$, définie par la Formule 3.21, peut également être évaluée plus finement en considérant seulement les modes atteignables à partir du mode m_k $(m_k, \delta_k) \in \zeta$. De même, les modes atteignables à partir du mode m_k seraient les seuls à prendre en compte dans l'estimation de la fonction $df_M(\Delta_3)$ de l'Equation 3.20.

Aucune tâche dans un module n'est active aux instants correspondant à l'hyperpériode de son mode courant d'exécution. Si les traces d'exécution des différents modules traversent simultanément des instants correspondant à leurs hyperpériodes, le processeur est inoccupé. L'analyse d'ordonnançabilité pour ces traces peut donc s'arrêter à ce point.

5.2.6 Réutilisation d'analyses

Le développement d'un système peut nécessiter plusieurs itérations avant le déploiement de sa version opérationnelle. Chaque itération apporte des modifications à l'itération précédente (des tâches, des modes, des modules sont ajoutés ou supprimés, leurs paramètres sont modifiés). Les résultats de l'analyse d'ordonnançabilité de l'itération antérieure peuvent être réutilisés lors de l'analyse d'ordonnançabilité de l'itération courante. Les parties du système qui ont subi des modifications peuvent nécessiter une nouvelle analyse mais les résultats antérieurs peuvent être utilisés pour celles qui sont restées intactes. De plus, certaines modifications, comme la suppression d'une tâche, n'affectent pas l'ordonnançabilité du système. Il serait intéressant de définir ces modifications afin d'effectuer analyse d'ordonnançabilité en se basant sur les résultats des analyses précédentes. Le temps d'exécution de l'analyse menée de cette façon pourrait se voir d'autant diminué.

5.2.7 Introduction de politiques à priorité fixe

Comme discuté en section 2.7, le test d'ordonnançabilité proposé par Farcas [59] sous des politiques à priorité fixe pour *TDL* est applicable aussi pour *E-TDL*. Le modèle de tâche *E-TDL*, en introduisant des décalages, augmente cependant le pessimisme du test. Les éléments d'analyse de configurations parallèles (section 3.6) peuvent être intégrés au test afin de réduire ce pessimisme.

L'instant critique n'existe pas toujours pour les tâches asynchrones (présentant des décalages). Dans ce cas, les algorithmes *Rate Monotonic* (section 2.3.1) et *Deadline Monotonic* (section 2.3.2) ne sont pas optimaux [17] et par conséquent, d'autres méthodes d'attribution

des priorités doivent être considérées. L'application de *l'algorithme d'Audsley* devrait en particulier être étudiée [16].

5.2.8 Extension à d'autres modèles d'allocations des ressources

Les ressources sont allouées aux modules *E-TDL* selon le modèle *périodique* (*Periodic Resource Model* [156], voir section 3.8). Cependant, il existe d'autres modèles applicables à *E-TDL* et permettant de partager les ressources de manière plus efficace.

Le modèle $\Omega(\Pi, \Theta, \Delta)$ *périodique avec échéance* (*Explicit Deadline Periodic*, *EDP*) [57] est une extension du modèle *périodique* $\Gamma(\Pi, \Theta)$. Les ressources Θ sont accessibles par période Π pendant une durée Δ à partir du début de la période courante. Le schéma d'allocation réalisé selon ce modèle utilise moins de ressources que le modèle périodique si la demande de temps processeur au sein d'un composant est bornée à un court intervalle de temps situé avant la fin de la période. En ajoutant des décalages à ce modèle, les demandes arrivant un certain temps après le début de la période pourraient être servies sans devoir surestimer la capacité du modèle. Ce modèle d'allocation des ressources pourrait également s'adapter aux changement de mode en acceptant des réservations sur plusieurs intervalles, chacun caractéristique d'un mode particulier, en ne délivrant les ressources seulement que dans l'intervalle caractérisant le mode en cours d'exécution.

5.2.9 Vers des architectures multiprocesseurs

La dernière décennie a vu une tendance croissante à l'utilisation d'architectures multiprocesseurs. Les résultats obtenus dans cette thèse mériteraient d'être étendus à ces architectures. Deux approches sont identifiées dans la littérature (section 2.2) à propos de l'ordonnancement au sein de ces architectures : l'ordonnancement par partitionnement et l'ordonnancement global [22].

Lorsque l'ordonnancement est réalisé par partitionnement, une tâche s'exécute toujours sur le même processeur. Le problème consiste donc à répartir au préalable les tâches de tous les modules d'un système *E-TDL* sur les processeurs disponibles en garantissant la faisabilité du système. Ce problème est analogue à celui du *bin packing* et fait partie de la classe des problèmes NP-difficiles [55]. Une fois résolu le problème de répartition des tâches sur les processeurs, l'ordonnançabilité des tâches s'exécutant sur chaque processeur peut être analysée à l'aide des méthodes proposées pour le monoprocesseur.

Lorsque l'ordonnancement est global, l'exécution d'une tâche n'est pas liée à un processeur particulier et le problème d'ordonnançabilité doit tenir compte des migrations de tâches entre processeurs. Plusieurs travaux traitent de ce problème en considérant les algorithmes d'ordonnancement à priorités fixes [32, 55] ainsi qu'à priorités dynamiques [73, 32, 55]. Le modèle de tâche à trois paramètres (C_i, D_i, T_i) est celui qui est le plus souvent utilisé.

La vérification d'ordonnançabilité d'ensembles de tâches asynchrones (ϕ_i, C_i, D_i, T_i)

s'appuie sur des techniques de parcours exhaustif. Le problème d'ordonnançabilité d'un système *E-TDL* pourrait être abordé par la mise en œuvre des mécanismes de changement de mode au sein de la boîte à outil *SchedMCore* [52, 53]. Une nouvelle borne de faisabilité tenant compte des changements de modes devrait être déterminée (des bornes pour les cas synchrone et asynchrone sans changement de mode sont données par Cucu-Grosjean et Goossens [54] ainsi que par Grolleau, Goossens et Cucu-Grosjean [78]).

5.3 Conclusions

Le but de l'ensemble de ces travaux est de proposer un langage de description temporelle pour des systèmes temps-réel et d'établir les conditions de leur ordonnançabilité sous l'algorithme *Earliest Deadline First*.

Le langage proposé, *Extended Timing Definition Language* (*E-TDL*), étend les langages déjà existant en introduisant un nouveau modèle de tâche. Ce modèle inclut quatre paramètres : phase Φ, pire temps d'exécution C, temps d'exécution logique TEL et période T. Il permet de modéliser des schémas d'exécution de tâches complexes et d'exprimer sans ambiguïté le temps de traitement nécessaire à leur exécution.

L'introduction de ce nouveau modèle de tâche requiert de revisiter en particulier le problème de l'ordonnançabilité des tâches, pour l'algorithme *Earliest Deadline First*. Le modèle de tâche considéré dans les travaux antérieurs est celui d'un modèle de tâche périodique à échéance sur requêtes. Il permet l'analyse de l'ordonnançabilité d'un ensemble de tâches en se fondant sur leur taux d'utilisation. Cette approche n'est plus applicable dans le modèle de tâche introduit par *E-TDL*. Cette thèse propose et développe une analyse basée sur la *fonction de demande* pour des ensembles de tâches décrites en *E-TDL* et s'exécutant en contexte *monoprocesseur*. Une condition nécessaire et suffisante est obtenue au travers d'une analyse précise des intervalles séparant les activations de tâches au sein de différents modules pouvant changer de mode à des instants prédéfinis. Une borne sur la longueur des intervalles devant être vérifiés est analytiquement déterminée.

Sur la base de ces résultats théoriques un outil permettant de vérifier l'ordonnançabilité de systèmes modélisés en *E-TDL* a été implanté en *Python*. La conception de l'outil s'appuie sur plusieurs algorithmes proposés pour :

- parcourir le graphe de changements de modes et trouver les instants de démarrage de chacun des modes ;
- positionner les instants d'activation des tâches au sein de différents modules et exhiber ainsi toutes les configurations dans lesquelles le système peut s'exécuter ;
- calculer la demande maximale faite par chacune des traces d'exécution d'un module *E-TDL*.

L'ordonnançabilité est d'abord vérifiée dans chaque intervalle avec un test approché de complexité moindre que celle du test exact. Le test exact est appliqué seulement si le test

approché ne parvient pas à garantir l'ordonnançabilité. Il résulte de l'attention apportée à limiter la complexité de l'analyse que sa performance est meilleure que celle du simulateur présenté dans ce mémoire.

De futurs possibles travaux pourraient viser l'assouplissement des règles de changement de mode de manière à rendre le système plus réactif. De nouveaux protocoles ainsi qu'une partie des résultats nécessaires à leur mise en œuvre sont proposés.

Plus généralement, les travaux réalisés dans cette thèse contribuent à changer les processus selon lesquels les applicatifs embarqués temps réel sont conçus, développés, mis au point et maintenus. La difficulté à laquelle ces processus sont confrontés, dans les approches traditionnelles, découle du fait que les différents concepts de la programmation temps réel, les aspects fonctionnel et temporel des applications temps réel embarqués ainsi que l'ensemble des caractéristiques de leur plateforme d'exécution, sont mêlés et doivent être traités ensemble.

Les mêmes applications peuvent se comporter différemment selon le matériel et la politique d'ordonnancement utilisés. La multiplicité de ces comportements complique les phases de tests et de certification dont l'analyse d'ordonnançabilité n'est souvent qu'un des plusieurs objectifs. Le langage de description temporelle *E-TDL* permet d'exprimer un certain nombre d'exigences ou de contraintes relatives à l'exécution des applications en temps réel séparément des fonctionnalités et sans considérer une architecture matérielle précise. Avec *E-TDL*, le développement du logiciel embarqué temps réel devient plus facile à maîtriser car il fait appel à trois composantes indépendantes associées au code fonctionnel, à la gestion du temps et à la machine d'exécution physique. Le modèle d'exécution logique des tâches impose que leurs dates de lecture d'entrées ainsi que leurs dates d'écriture de sorties soient les mêmes sur toutes les architectures rendant ainsi le comportement des tâches identique sur différentes plateformes d'exécution. Les dernières décennies font apparaître, face à la croissance de la part du logiciel dans les systèmes, une tendance à décomposer les applications en plusieurs composants de manière à pouvoir les développer indépendamment en s'abstrayant des éléments de bas niveau de détail. L'approche proposée dans *E-TDL* s'inscrit dans cette veine. Il ne s'agit pas seulement de l'écriture modulaire des programmes. La gestion du temps, qui est une ressource dépendante de la plateforme d'exécution, cruciale dans les système temps réel, est externalisée, dans les processus de conception et de développement, pour assurer une séparation claire et nette entre la part fonctionnelle et la part des exigences temporelles. Le programmeur peut ainsi se focaliser sur la programmation des fonctions dont l'exécution, décrite dans une spécification de ses contraintes temporelles, est guidée selon un schéma produit par le compilateur. Celui-ci est en charge d'adapter la politique d'ordonnancement et d'établir le protocole de communication commun conforme à la définition du système rédigée en *E-TDL*.

Bibliographie

[1] TDL - Timing Definition Language Specification. Technical report, www.chrona.com/fileadmin/user_upload/downloads/TDLReport.pdf, June 2011.

[2] L. Abeni and G. Buttazzo. Resource Reservation in Dynamic Real-Time Systems. *Real-Time Systems*, 27(2) :123–167, July 2004.

[3] L. Abeni, G. Buttazzo, S. Superiore, and S. Anna. Integrating Multimedia Applications in Hard Real-Time Systems. In *In Proceedings of the 19th IEEE Real-time Systems Symposium*, pages 4–13, 1998.

[4] M. Ahmed and N. Fisher. Tractable Schedulability Analysis and Resource Allocation for Real-Time Multimodal Systems. *ACM Trans. Embedded Comput. Syst.*, 13(2s) :65, 2014.

[5] K. Albers et F. Slomka. An Event Etream Driven Approximation for The Analysis of Real-Rime Systems. In *In Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 187–195. IEEE Computer Society Press, 2004.

[6] K. Albers et F. Slomka. Efficient Feasibility Analysis for Real-Time Systems with EDF scheduling. In *Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*, DATE '05, pages 492–497, Washington, DC, USA, 2005. IEEE Computer Society.

[7] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.

[8] Rajeev Alur and Gera Weiss. Rtcomposer : a framework for real-time components with scheduling interfaces. In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 159–168, 2008.

[9] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES - A tool for modelling and implementation of embedded systems. In *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, pages 460–464, 2002.

[10] T. Amnell, E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. TIMES : A tool for schedulability analysis and code generation of real-time systems. In *Formal Modeling and Analysis of Timed Systems : First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers*, pages 60–72, 2003.

[11] B. Andersson. Uniprocessor EDF Scheduling with Mode Change. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, pages 572–577, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] R. V. Aroca, G. Caurin, and S. Carlos-SP-Brasil. A real time operating systems (rtos) comparison. In *Workshop de Sistemas Operacionais (Operating Systems)(WSO'2009)*, 2009.

[13] N. Audsley, A. Burns, M. Richardson, K. Tindell, et A. J. Wellings. Applying New Scheduling Theory to Static Priority Pre-emptive scheduling. *Software Engineering Journal*, 8 :284–292, 1993.

[14] Neil C. Audsley. Deadline Monotonic Scheduling, 1990.

[15] N. Audsley, A. Burns, M. F. Richardson, et A. J. Wellings. Hard Real-Time Scheduling : The Deadline-Monotonic Approach. In *in Proc. IEEE Workshop on Real-Time Operating Systems and Software*, pages 133–137, 1991.

[16] N.C. Audsley. *Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times.* Technical report. University of York, Department of Computer Science, 1991.

[17] Neil C. Audsley, Ken Tindell, and Alan Burns. The end of the line for static cyclic scheduling ? In *Fifth Euromicro Workshop on Real-Time Systems, RTS 1993, Oulu, Finland, June 22-24, 1993. Proceedings.*, pages 36–41, 1993.

[18] Bruno D'Ausbourg et Frédéric Boniol. Le langage TDLBono. Un langage pour les systèmes dirigés par le temps. Technical report, Département Traitement de l'Information. ONERA, Juillet 2010.

[19] François Baccelli, Guy Cohen, Geert Jan Olsder, and Jean-Pierre Quadrat. *Synchronization and linearity : an algebra for discrete event systems.* Wiley series in probability and mathematical statistics. J. Wiley & Sons, Chichester, New York, 1992.

[20] T. P. Baker. Stack-Based Scheduling for Real-Time Processes. *Real-Time Syst.*, 3(1) :67–99, Apr. 1991.

[21] S. K. Baruah. Resource Sharing in EDF-Scheduled Systems : A Closer Look. In *RTSS*, pages 379–387. IEEE Computer Society, 2006.

[22] S. K. Baruah, M. Bertogna, and G. C. Buttazzo. *Multiprocessor Scheduling for Real-Time Systems.* Embedded Systems. Springer, 2015.

[23] S. K. Baruah et A. Burns. Sustainable Scheduling Analysis. In *Proceedings of the 27th IEEE Real-Time Systems Symposium (RTSS 2006), 5-8 December 2006, Rio de Janeiro, Brazil*, pages 159–168, 2006.

[24] Sanjoy K. Baruah, Rodney R. Howell, et Louis Rosier. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic, Real-Time Tasks on One Processor. *Real-Time Systems*, 2 :301–324, 1990.

[25] S. K. Baruah. Feasibility Analysis of Recurring Branching Tasks. In *ECRTS*, pages 138–145, 1998.

[26] S. Baruah. Preemptive uniprocessor scheduling of non-cyclic gmf task systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, pages 195–202, Aug 2010.

[27] Sanjoy Baruah, Deji Chen, Sergey Gorinsky, et Aloysius Mok. Generalized Multiframe Tasks. *Real-Time Syst.*, 17(1) :5–22, July 1999.

[28] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on uppaal. In M. Bernardo and F. Corradini, editors, *International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004. Revised Lectures*, volume 3185 of *Lecture Notes in Computer Science*, pages 200–237. Springer Verlag, 2004.

[29] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1) :64–83, Jan 2003.

[30] Gérard Berry and Laurent Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*, pages 389–448, 1984.

[31] Gérard Berry and Georges Gonthier. The esterel synchronous programming language : Design, semantics, implementation. *Science of computer programming*, 19(2) :87–152, 1992.

[32] M. Bertogna, M. Cirinei, and G. Lipari. Schedulability analysis of global scheduling algorithms on multiprocessor platforms. *Parallel and Distributed Systems, IEEE Transactions on*, 20(4) :553–566, April 2009.

[33] E. Bini, M. Bertogna, and S. K. Baruah. Virtual Multiprocessor Platforms : Specification and Use. In T. P. Baker, editor, *RTSS*, pages 437–446. IEEE Computer Society, 2009.

[34] Enrico Bini, Giorgio C. Buttazzo, et Giuseppe M. Buttazzo. A Hyperbolic Bound For The Rate Monotonic Algorithm, 2001.

[35] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt, et J Weglarz. *Handbook on Scheduling. From Theory to Applications.* Springer, 2007.

[36] O. Bordellès and V. Bordellès. *Arithmetic Tales.* Universitext Series. Springer, 2012.

[37] G. Buttazzo, G. Lipari, et L. Abeni. Elastic Task Model For Adaptive Rate Control. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 286 –295, Dec. 1998.

[38] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems : Predictable Scheduling Algorithms and Applications*. Springer, second edition, 2005.

[39] Giorgio C. Buttazzo. Rate Monotonic vs. EDF : Judgment Day. *Real-Time Syst.*, 29(1) :5–26, Jan. 2005.

[40] G. C. Buttazzo et L. Abeni. Adaptive Workload Management Through Elastic Scheduling. *Real-Time Systems*, 23(1-2) :7–24, 2002.

[41] Giorgio Buttazzo and Giuseppe Lipari. Ptask : An educational C library for programming real-time systems on linux. In *Proceedings of the 18th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'13)*, Cagliari, Italy, September 2013. IEEE Industrial Electronics Society.

[42] G. C. Buttazzo, G. Lipari, M. Caccamo, et L. Abeni. Elastic Scheduling For Flexible Workload Management. *IEEE Trans. Computers*, 51(3) :289–302, 2002.

[43] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre : A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 178–188, 1987.

[44] S. Chakraborty, S. Kunzli, et L. Thiele. Approximate Schedulability Analysis. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 159 – 168, 2002.

[45] Younès Chandarli. *Gestion de l'énergie renouvelable et ordonnancement temps réel dans les systèmes embarqués*. Theses, Université Paris-Est, December 2014.

[46] Younès Chandarli, Frédéric Fauberteau, Damien Masson, Serge Midonnet, and Manar Qamhieh. YARTISS : A Tool to Visualize, Test, Compare and Evaluate Real-Time Scheduling Algorithms. In *WATERS 2012*, pages 21–26, Italy, July 2012.

[47] Maxime Chéramy. *Study and evaluation of multiprocessor real-time scheduling policies*. Theses, Institut National des Sciences Appliquées de Toulouse (INSA Toulouse), December 2014.

[48] Maxime Chéramy, Anne-Marie Déplanche, and Pierre-Emmanuel Hladik. Simulation of real-time multiprocessor scheduling with overheads. In *SIMULTECH 2013 - Proceedings of the 3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications, Reykjavík, Iceland, 29-31 July, 2013*, pages 5–14, 2013.

[49] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. SimSo : A Simulation Tool to Evaluate Real-Time Multiprocessor Scheduling Algorithms. In *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, page 6 p., Madrid, Spain, July 2014.

[50] Maxime Chéramy, Pierre-Emmanuel Hladik, Anne-Marie Déplanche, and Sébastien Dubé. Simulation of Real-Time Scheduling with Various Execution Time Models. In *9th IEEE International Symposium on Industrial Embedded Systems (SIES)*, Pise, Italy, June 2014. Presented during the Work-in-Progress session (WiP session).

[51] A. Choquet-Geniet and E. Grolleau. Minimal schedulability interval for real time systems of periodic tasks with offsets. *Theoretical of Computer Sciences*, 2004 :117–134, 2004.

[52] Mikel Cordovilla. *Environnement de développement d'applications multipériodiques sur plateforme multicoeur. La boîte à outil SchedMCore.* Theses, Université Paul Sabatier - Toulouse III, April 2012.

[53] Mikel Cordovilla, Frédéric Boniol, Eric Noulard, and Claire Pagetti. Multiprocessor schedulability analyser. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), TaiChung, Taiwan, March 21 - 24, 2011*, pages 735–741, 2011.

[54] Liliana Cucu and Joël Goossens. Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems. In Rudy Lauwereins and Jan Madsen, editors, *DATE*, page 1635–1640. ACM, 2007.

[55] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4) :35 :1–35 :44, October 2011.

[56] P. Derler and S. Resmerita. Flexible Static Scheduling of Software with Logical Execution Time Constraints. In *10th IEEE International Conference on Computer and Information Technology, CIT 2010, Bradford, West Yorkshire, UK, June 29-July 1, 2010*, pages 1719–1726, 2010.

[57] A. Easwaran, M. Anand, and I. Lee. Compositional Analysis Framework using EDP Resource Models. In *Proceedings of the 28th IEEE International Real-Time Systems Symposium*, RTSS '07, pages 129–138, Washington, DC, USA, 2007. IEEE Computer Society

[58] C. Farcas and W. Pree. Virtual execution environment for real-time tdl components. In *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, pages 93–100, Sept 2007.

[59] E. Farcas. *Scheduling Multi-Mode Real-Time Distributed Components.* PhD Thesis, Department of Computer Science, University of Salzburg, July 2006.

[60] E. Farcas, C. Farcas, W. Pree, and J. Templ. Transparent Distribution of Real-Time Components Based on Logical Execution Time. In *LCTES*, pages 31–39, 2005.

[61] Peter H Feiler, David P Gluch, and John J Hudak. The architecture analysis & design language (aadl) : An introduction. Technical report, DTIC Document, 2006.

[62] N. Fisher, M. Ahmed, and P. M. Hettiarachchi. Open Problems in Multi-Modal Scheduling Theory for Thermal-Resilient Multicore Systems. In *Proceedings of the 5th Real-Time Scheduling Open Problems Seminar, RTSOPS 2014, Madrid, Spain, 8 July 2014.*

[63] N. Fisher and M. Ahmed. Tractable Real-Time Schedulability Analysis for Mode Changes under Temporal Isolation. In *ESTImedia*, pages 130–139, 2011.

[64] G. Fohler. Changing Operational Modes in the Context of Pre Run-Time Scheduling, 1993.

[65] J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. PhD thesis, Institut Supérieur de l'Aéronautique et de l'Espace, 2009.

[66] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. A multi-periodic synchronous data-flow language. In *11th IEEE High Assurance Systems Engineering Symposium, HASE 2008, Nanjing, China, December 3 - 5, 2008*, pages 251–260, 2008.

[67] Paolo Gai, Enrico Bini, Giuseppe Lipari, Marco Di Natale, and Luca Abeni. Architecture for a portable open source real time kernel environment. In *In Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, 2000.

[68] Paolo Gai, Giuseppe Lipari, and Marco Di Natale. Stack size minimization for embedded real-time systems-on-a-chip. *Design Automation for Embedded Systems*, 7(1-2) :53–87, 2002.

[69] J. V. Z. Gathen et J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, New York, NY, USA, 2 edition, 2003.

[70] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-Time UniProcessor Scheduling. Research Report RR-2966, 1996. Projet REFLECS.

[71] Arkadeb Ghosal. Expressing Giotto in xGiotto and Related Observations on Schedulability Analysis.

[72] Arkadeb Ghosal, Thomas A. Henzinger, Christoph M. Kirsch, and Marco A. A. Sanvido. Event-Driven Programming with Logical Execution Times. In *Hybrid Systems : Computation and Control, 7th International Workshop, HSCC 2004, Philadelphia, PA, USA, March 25-27, 2004, Proceedings*, pages 357–371, 2004.

[73] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3) :187–205, 2003.

[74] J. Goossens. Scheduling of Offset Free Systems. *Real-Time Systems*, 24(2) :239–258, 2003.

[75] Grolleau Emmanuel. Tutorial on real-time scheduling. In *Ecole d'été temps réel, ETR'07*, Nantes, 2007.

[76] E. Grolleau and A. Choquet-Geniet. Cyclicité des ordonnancements de systèmes de tâches périodiques différées. In Teknea, editor, *Real-Time Systems, RTS'2000*, Paris, 2000.

[77] E. Grolleau, A. Choquet-Geniet, and F. Cottet. Cyclicité des ordonnancements au plus tôt des systèmes de tâches temps réel. In *Rencontres Francophones du Parallélisme, RenPar'10*, pages 39–42, Strasbourg, 1998.

[78] Grolleau Emmanuel, Goossens Joël, and Cucu-Grosjean Liliana. On the periodic behavior of real-time schedulers on identical multiprocessor platforms. Technical Report arXiv 1305.3849, 2013.

[79] Q. Guangming. An earlier time for inserting and/or accelerating tasks. *Real-Time Systems*, 41(3) :181–194, 2009.

[80] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[81] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9) :1305–1320, Sep 1991.

[82] D. Harel and A. Pnueli. Logics and models of concurrent systems. chapter On the Development of Reactive Systems, pages 477–498. Springer-Verlag New York, Inc., New York, NY, USA, 1985.

[83] T. A. Henzinger. Composable Code Generation for Distributed Giotto. In *LCTES*, pages 21–30, 2005.

[84] T.A. Henzinger, C.M. Kirsch, E. Marques, and A. Sokolova. Distributed, modular htl. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 171–180, Dec 2009.

[85] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Embedded Control Systems Development with Giotto. In *LCTES/OM*, pages 64–72, 2001.

[86] T. A. Henzinger, B. Horowitz, and C. M. Kirsch. Giotto : A Time-Triggered Language For Embedded Programming. In *EMSOFT*, pages 166–184, 2001.

[87] T. A. Henzinger and C. M. Kirsch. The embedded machine : Predictable, portable real-time code. *ACM Trans. Program. Lang. Syst.*, 29(6), Oct. 2007.

[88] Thomas A. Henzinger, Christoph M. Kirsch, Rupak Majumdar, and Slobodan Matic. Time-Safety Checking for Embedded Programs. In *Embedded Software, Second International Conference, EMSOFT 2002, Grenoble, France, October 7-9, 2002, Proceedings*, pages 76–92, 2002.

[89] P.M. Hettiarachchi, N. Fisher, M. Ahmed, Le Yi Wang, Shinan Wang, and Weisong Shi. The design and analysis of thermal-resilient hard-real-time systems. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 67–76, April 2012.

[90] D. Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[91] W. A. Horn. *Some Simple Scheduling Algorithms.* Naval Research Logistics Quarterly, 1974.

[92] James R. Jackson. *Scheduling a Production Line to Minimize Maximum Tardiness*. University of California, 1955.

[93] D. Jelkmann, K. Albers, et F. Slomka. Improved Feasibility Tests for Asynchronous Real-Time Periodic Task Sets. In C. Haubelt and J. Teich, editors, *MBMV*, pages 69–78. Shaker, 2007

[94] M. Joseph et P. K. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5) :390–395, 1986.

[95] R. M. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3) :309 – 311, 1978.

[96] J. Kleinberg and E. Tardos. *Algorithm Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005.

[97] Tomasz Kloda, Bruno d'Ausbourg, and Luca Santinelli. EDF Schedulability Analysis for an Extended Timing Definition Language. In *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems, SIES 2014, Pisa, Italy, June 18-20, 2014*, pages 30–40, 2014.

[98] Tomasz Kloda, Bruno D'Ausbourg, and Luca Santinelli. Towards a More Flexible Timing Definition Language. In *12th International Workshop Quantitative Aspects of Programming Languages and Systems-at ETAPS 2014*, 2014.

[99] Tomasz Kloda, Bruno d'Ausbourg, and Luca Santinelli. Towards EDF Schedulability Analysis of an Extended Timing Definition Language. *SIGBED Review*, 11(3) :44–49, 2014.

[100] Tomasz Kloda, Bruno d'Ausbourg, and Luca Santinelli. Schedulability Analysis with an Extended Timing Definition Language. In *Real Time Scheduling Open Problems Seminar*, page 20, 2013.

[101] D. E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.) : Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[102] S. Koenig, M. Likhachev, Y. Liu, and D. Furcy. Incremental heuristic search in ai. *AI Mag.*, 25(2) :99–112, June 2004.

[103] H. Kopetz. Event-triggered versus time-triggered real-time systems. In *Proceedings of the International Workshop on Operating Systems of the 90s and Beyond*, pages 87–101, London, UK, UK, 1991. Springer-Verlag.

[104] J.-Y. Le Boudec and P. Thiran. *Network Calculus : A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

[105] John P. Lehoczky, Lui Sha, et Y. Ding. The Rate Monotonic Scheduling Algorithm : Exact Characterization and Average Case Behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[106] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. In *Proceedings of the 8th IEEE Real-Time Systems Symposium (RTSS '87), December 1-3, 1987, San Jose, Califorinia, USA*, pages 261–270, 1987.

[107] J. Leung et M. Merrill. A Note on The Preemptive Scheduling of Periodic, Real-Time Tasks. *Information Processing Letters*, 11 :115–118, 1980.

[108] Joseph Yuk-Tong Leung et Jennifer Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, 2(4) :237–250, 1982.

[109] H. Lin. The development of software for ballistic-missile defense. *Sci. Am.*, 253(6) :46–53, Dec. 1985.

[110] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *Real-Time Systems Symposium, 2000. Proceedings. The 21st IEEE*, pages 217–226, 2000.

[111] Giuseppe Lipari, Paolo Gai, Michael Trimarchi, Giacomo Guidi, and Paolo Ancilotti. A hierarchical framework for component-based real-time systems. In Ivica Crnkovic, JudithA. Stafford, HeinzW. Schmidt, and Kurt Wallnau, editors, *Component-Based Software Engineering*, volume 3054 of *Lecture Notes in Computer Science*, pages 209–216. Springer Berlin Heidelberg, 2004.

[112] C. L. Liu et James W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, 20(1) :46–61, January 1973.

[113] Jane W. S. Liu. *Real Time Systems*. Prentice-Hall, Inc., 2000.

[114] N. F. Martinek et W. Pohlmann. Mode Switching in GIA – An Ada Based Real-Time Framework. Department of Scientific Computing, University of Salzburg.

[115] Aloysius Ka-Lau Mok. *Fundamental Design Problems of Distributed Systems for The Hard-Real-Time Environment*. PhD thesis, Massachusetts Institute of Technology, May 1983.

[116] Aloysius K. Mok et Deji Chen. A Multiframe Model for Real-Time Tasks. *IEEE Trans. Software Eng.*, 23(10) :635–645, 1997.

[117] A. Molano, K. Juvva, and R. Rajkumar. Real-time filesystems. guaranteeing timing constraints for disk accesses in rt-mach. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 155–165, Dec 1997.

[118] Noël Tchidjo Moyo. *Architecture logicielle et méthodologie de conception embarquée sous contraintes temps réel pour la radio logicielle*. Theses, Université Rennes 1, April 2011.

[119] N.T. Moyo, E. Nicollet, F. Lafaye, and C. Moy. On schedulability analysis of non-cyclic generalized multiframe tasks. In *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*, pages 271–278, July 2010.

[120] V. Nélis. *Energy-Aware Real-Time Scheduling in Embedded Multiprocessor Systems*. PhD thesis, U.L.B., 2011.

[121] V. Nélis, B. Andersson, J. Marinho, et S. M. Petters. Global-edf scheduling of multimode real-time systems considering mode independent tasks. In *23rd Euromicro Conference on Real-Time Systems, ECRTS 2011, Porto, Portugal, 5-8 July, 2011*, pages 205–214, 2011.

[122] V. Nélis and J. Goossens. Mode Change Protocol for Multi-Mode Real-Time Systems upon Identical Multiprocessors. *CoRR*, abs/0809.5238, 2008.

[123] V. Nélis, J. Goossens, and B. Andersson. Two Protocols for Scheduling Multi-Mode Real-Time Systems upon Identical Multiprocessor Platforms. In *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland, July 1-3, 2009*, pages 151–160, 2009.

[124] Yassine Ouhammou. *Model-based Framework for Using Advanced Scheduling Theory in Real-Time Systems Design*. PhD thesis, dec 2013.

[125] Yassine Ouhammou, Emmanuel Grolleau, and Jerome Hugues. Mapping aadl models to a repository of multiple schedulability analysis techniques. In *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2013 IEEE 16th International Symposium on*, pages 1–8. IEEE, 2013.

[126] Yassine Ouhammou, Emmanuel Grolleau, Michael Richard, Pascal Richard, and Frédéric Madiot. Mosart framework : a collaborative tool for modeling and analyzing embedded real-time systems. In Springer Verlag, editor, *Complex Systems Design & Management*, page 12, 2014.

[127] José Carlos Palencia and M González Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*, pages 26–37. IEEE, 1998.

[128] Luigi Palopoli, Giuseppe Lipari, Luca Abeni, Marco Di Natale, Paolo Ancilotti, and Fabio Conticelli. A tool for simulation and fast prototyping of embedded control systems. In *Proceedings of The Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES 2001), June 22-23, 2001 / The Workshop on Optimization of Middleware and Distributed Systems (OM 2001), June 18, 2001, Snowbird, Utah, USA*, pages 73–81, 2001.

[129] Luigi Palopoli, Giuseppe Lipari, Gerardo Lamastra, Luca Abeni, Gabriele Bolognini, and Paolo Ancilotti. An object-oriented tool for simulating distributed real-time control systems. *Softw., Pract. Exper.*, 32(9) :907–932, 2002.

[130] Paulo Pedro et Alan Burns. Schedulability Analysis for Mode Changes in Flexible Real-Time systems. In *10th Euromicro Workshop on Real-Time Systems'98*, pages 172–179, 1998.

[131] Paulo Sérgio M. Pedro. *Schedulability of Mode Changes in Flexible Real-Time Distributed Systems*. PhD thesis, University of York, september 1999.

[132] R. Pellizzoni. *Efficient Feasibility Analysis of Real-Time Asynchronous Task Sets*. PhD thesis, Università Di Pisa and Scuola Superiore S. Anna, 2003.

[133] R. Pellizzoni and G. Lipari. Feasibility Analysis of Real-Time Periodic Tasks with Offsets. *Real-Time Systems*, 30(1-2) :105–128, 2005.

[134] S. Perathoner, N. Stoimenov, and L. Thiele. Reliable Mode Changes in Real-Time Systems with Fixed Priority or EDF Scheduling. TIK Report 292, Computer Engineering and Networks Laboratory, ETH Zurich, September 2008.

[135] L. T. Phan, S. Chakraborty, and I. Lee. Timing Analysis of Mixed Time/Event-Triggered Multi-Mode Systems. In *Real-Time Systems Symposium, 2009, RTSS 2009. 30th IEEE*, pages 271–280. IEEE, 2009.

[136] L. T. X. Phan, S. Chakraborty, and P. S. Thiagarajan. A Multi-Mode Real-Time Calculus. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, pages 59–69, 2008.

[137] L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional Analysis of Multi-Mode Systems. In *ECRTS*, pages 197–206, 2010.

[138] L. T. X. Phan, I. Lee, and O. Sokolsky. Compositional Analysis of Multi-Mode Systems. In *22nd Euromicro Conference on Real-Time Systems, ECRTS 2010, Brussels, Belgium, July 6-9, 2010*, pages 197–206, 2010.

[139] Linh T. X. Phan, Insup Lee, and Oleg Sokolsky. A semantic framework for mode change protocols. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 91–100. IEEE Computer Society, 2011.

[140] W. Pree and J. Templ. Modeling with the Timing Definition Language (TDL). In *ASWSD*, pages 133–144, 2006.

[141] Ahmed Rahni, Emmanuel Grolleau, and Michael Richard. Feasibility analysis of non-concrete real-time transactions with edf. In *16th International Conference on Real-Time and Network Systems RTNS2008 Rennes France*, 2008.

[142] J. Ramírez-Alfonsín. Complexity of the Frobenius problem. *Combinatorica*, 16 :143–147, 1996.

[143] Jorge Real et Alfons Crespo. Mode Change Protocols for Real-Time Systems : A Survey and a New Proposal. *Real-Time Syst.*, 26(2) :161–197, March 2004.

[144] Jorge Vicente Real Sáez. *Protocolos de Cambio de Modo Para Sistemas de Tiempo Real*. PhD thesis, Universitat Politècnica de València, 2000.

[145] S. Resmerita and P. Derler. Flexible Scheduling of Predictable Software with Logical Execution Time Constraints. In *Workshop on Reconciling Performance with Predictability (RePP)*, oct 2009.

[146] Pascal Richard. Analyse du temps de réponse des systèmes temps réel, 2003.

[147] Pascal Richard et Frédéric Ridouard. *Ordonnancement temps réel monoprocesseur*, Chapitre 1. Hermès, 2006.

[148] I. Ripoll, A. Crespo, et A. K. Mok. Improvement in Feasibility Testing For Real-Time Tasks. *Real-Time Systems*, 11(1) :19–39, 1996.

[149] L. Santinelli, G. C. Buttazzo, and E. Bini. Multi-Moded Resource Reservations. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 37–46, 2011.

[150] Marco AA Sanvido, Arkadeb Ghosal, and Thomas A Henzinger. Xgiotto language report. Technical report, Computer Science Division, University of California, 2003.

[151] Lui Sha, Ragunathan Rajkumar, John Lehoczky, et Krithi Ramamritham. Mode Change Protocols for Priority-Driven Preemptive Scheduling. *Real-Time Systems*, 1 :243–264, 1988.

[152] L. Sha, R. Rajkumar, et J. P. Lehoczky. Priority Inheritance Protocols : An Approach to Real-Time Synchronization. *IEEE Trans. Comput.*, 39(9) :1175–1185, Sept. 1990.

[153] J. Shallit. The Frobenius Problem and Its Generalizations. In M. Ito and M. Toyama, editors, *Developments in Language Theory*, volume 5257 of *Lecture Notes in Computer Science*, pages 72–83. Springer Berlin Heidelberg, 2008.

[154] I. Shin, A. Easwaran, and I. Lee. Hierarchical Scheduling Framework for Virtual Clustering of Multiprocessors. In *20th Euromicro Conference on Real-Time Systems, ECRTS 2008, 2-4 July 2008, Prague, Czech Republic, Proceedings*, pages 181–190, 2008.

[155] Insik Shin and Insup Lee. Compositional Real-Time Scheduling Framework with Periodic Model. *ACM Trans. Embed. Comput. Syst.*, 7(3) :30 :1–30 :39, May 2008.

[156] Insik Shin and Insup Lee. Periodic Resource Model for Compositional Real-Time Guarantees. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003), 3-5 December 2003, Cancun, Mexico*, pages 2–13, 2003.

[157] F. Singhoff, J. Legrand, L. Nana, and L. Marcé. Cheddar : A flexible real time scheduling framework. *Ada Lett.*, XXIV(4) :1–8, November 2004.

[158] S. S. Skiena. *The Algorithm Design Manual*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[159] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic Task Scheduling for Hard Real-Time Systems. *Real-Time Systems*, 1(1) :27–60, 1989.

[160] M. Spuri. Analysis of Deadline Scheduled Real-Time Systems. Rapport de recherche, Instiut National de Recherche en Informatique et en Automatique, 1996.

[161] M. Spuri and G. Buttazzo. Scheduling Aperiodic Tasks in Dynamic Priority Systems. *Real-Time Systems*, 10 :179–210, 1996.

[162] M. Spuri and G. C. Buttazzo. Efficient Aperiodic Service under Earliest Deadline Scheduling. In *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS '94), San Juan, Puerto Rico, December 7-9, 1994*, pages 2–11, 1994.

[163] R. Stair and G. Reynolds. *Principles of Information Systems*. Course Technology Press, Boston, MA, United States, 9th edition, 2009.

[164] J. A. Stankovic. Misconceptions about real-time computing : A serious problem for next-generation systems. *Computer*, 21(10) :10–19, Oct. 1988.

[165] John A. Stankovic and K. Ramamritham, editors. *Tutorial : Hard Real-time Systems.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1989.

[166] M. Stigge, P. Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*, pages 71–80, April 2011.

[167] N. Stoimenov, S. Perathoner, et L. Thiele. Reliable Mode Changes in Real-Time Systems with Fixed Priority or EDF Scheduling. In *Proceedings of Design, Automation and Test in Europe, 2009 (DATE 09)*, pages 99–104, Nice, France, Apr 2009. IEEE.

[168] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments. *IEEE Trans. Comput.*, 44(1) :73–91, Jan. 1995.

[169] L. Thiele, S. Chakraborty, et M. Naedele. Real-Time Calculus for Scheduling Hard Real-Time Systems. In *The 27th Annual International Symposium on Computer Architecture(ISCA)*, volume 4, pages 101 –104 vol.4, 2000.

[170] K. Tindell. *Adding Time-offsets to Schedulability Analysis.* Technical report. University of York, Department of Computer Science, England, Jan. 1994.

[171] K. Tindell et A. Alonso. A Very Simple Protocol for Mode Changes in Priority Preemptive Systems. Technical report, Universidad Politécnica de Madrid, 1996.

[172] K. W. Tindell, A. Burns, et A. J. Wellings. Mode Changes in Priority Pre-emptively Scheduled Systems. In *Proceedings of the Real Time Systems Symposium*, pages 100–109, 1992.

[173] Richard Urunuela, Anne-Marie Déplanche, and Yvon Trinquet. STORM a simulation tool for real-time multiprocessor scheduling evaluation. In *Proceedings of 15th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2010, September 13-16, 2010, Bilbao, Spain*, pages 1–8, 2010.

[174] V. Yodaiken. The RTLinux Manifesto. In *Proc. of The 5th Linux Expo, Raleigh, NC*, March 1999.

[175] P. M. Yomsi, V. Nélis, et J. Goossens. Scheduling Multi-Mode Real-Time Systems upon Uniform Multiprocessor Platforms. *CoRR*, abs/1004.3687, 2010.

[176] E. Wandeler. *Modular Performance Analysis and Interface Based Design for Embedded Real-Time Systems.* PhD thesis, ETH Zurich, 2006.

[177] E. Wandeler, A. Maxiaguine, and L. Thiele. Quantitative Characterization of Event Streams in Analysis of Hard Real-Time Applications. *Real-Time Systems*, 29(2-3) :205–225, 2005.

[178] Haibo Zeng and Marco Di Natale. Computing periodic request functions to speed-up the analysis of non-cyclic task models. *Real-Time Systems*, pages 1–35, 2014.

[179] Haibo Zeng and Marco Di Natale. Outstanding paper award : Using max-plus algebra to improve the analysis of non-cyclic task models. In *25th Euromicro Conference on Real-Time Systems, ECRTS 2013, Paris, France, July 9-12, 2013*, pages 205–214, 2013.

[180] Fengxiang Zhang and A. Burns. Improvement to quick processor-demand analysis for edf-scheduled real-time systems. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 76–86, July 2009.

[181] Fengxiang Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *Computers, IEEE Transactions on*, 58(9) :1250–1258, Sept 2009.